NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

GUARANTEEING FAIR ACCESS FOR MULTIMEDIA TRAFFIC USING THE SMART ALGORITHM FOR MULTICAST ATM CONNECTIONS

by

Randolph R. Weekly

September 1999

Thesis Adviser: Murali Tummala
Co-Advisor: Robert E. Parker, Jr.

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1999	Master's Thesis

4. TITLE AND SUBTITLE
GUARANTEEING FAIR ACCESS FOR MULTIMEDIA TRAFFIC USING THE SMART ALGORITHM FOR MULTICAST ATM CONNECTIONS

5. FUNDING NUMBERS

6. AUTHOR(S)
Weekly, Randolph R.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The cornerstone of the US Navy's implementation of network centric warfare is the Information Technology for the 21st Century (IT-21) Intranet. The IT-21 Intranet relies on Asynchronous Transfer Mode (ATM) backbones for data transfer. An essential service provided by the IT-21 Intranet is video teleconferencing (VTC). However, the current ATM standard does not support the multipoint-to-multipoint multicast requirements for robust VTC applications.

This work builds upon the SMART algorithm, an existing protocol that provides support for ATM multipoint-to-multipoint multicast over a single virtual channel (VCC), and modifies it to ensure that each source has fair access to the VCC. Fair access is ensured to multiple sources through the use of a metric that takes into account the relative magnitude of each source's queue size and cell age. The improved SMART algorithm is modeled using the MIL3's Optimized Network Engineering Tool (OPNET). Fair access to each source is shown using both audio and video source models. VCC utilization is examined and shown to improve for the video case as the number of sources increase. The increased overhead due to the use of Resource Management (RM) cells is also examined.

14. SUBJECT TERMS
Video Teleconferencing, Asynchronous Transfer Mode, Multicast, Networks

15. NUMBER OF PAGES
142

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited.

GUARANTEEING FAIR ACCESS FOR MULTIMEDIA TRAFFIC USING THE SMART ALGORITHM FOR MULTICAST ATM CONNECTIONS

Randolph R. Weekly
Lieutenant Commander, United States Navy
B.S., University of Southern California, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

September 1999

Author: Randolph R. Weekly

Approved by: Murali Tummala, Thesis Advisor

Robert E. Parker Jr., Co-Advisor

Jeffrey B. Knorr, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The cornerstone of the US Navy's implementation of network centric warfare is the Information Technology for the 21st Century (IT-21) Intranet. The IT-21 Intranet relies on Asynchronous Transfer Mode (ATM) backbones for data transfer. An essential service provided by the IT-21 Intranet is video teleconferencing (VTC). However, the current ATM standard does not support the multipoint-to-multipoint multicast requirements for robust VTC applications.

This work builds upon the SMART algorithm, an existing protocol that provides support for ATM multipoint-to-multipoint multicast over a single virtual channel (VCC), and modifies it to ensure that each source has fair access to the VCC. Fair access is ensured to multiple sources through the use of a metric that takes into account the relative magnitude of each source's queue size and cell age. The improved SMART algorithm is modeled using the MIL3's Optimized Network Engineering Tool (OPNET). Fair access to each source is shown using both audio and video source models. VCC utilization is examined and shown to improve for the video case as the number of sources increase. The increased overhead due to the use of Resource Management (RM) cells is also examined.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS AND ABBREVIATIONS

AAL	ATM Adaptation Layer
ABR	Available Bit Rate
ATM	Asynchronous Transfer Mode
B-ISDN	Broadband Integrated Services Digital Network
BN	Backward Explicit Congestion Notification
CBR	Constant Bit Rate
CCR	Current Cell Rate
CI	Congestion Indicator
COTS	Commercial Off-The-Shelf Technology
CPCS	Common-part Convergence Sublayer
CRC	Cyclic Redundancy Check
ECR	Explicit Cell Rate
EHF	Extremely High Frequency
EOP	End of Packet
GFC	Generic Flow Control
ID	Identifier
ITU-T	International Telecommunication Union Telecommunication Standardization Sector
LAN	Local Area Network
LOS	Line of Sight
MCR	Minimum Cell Rate
MCS	Multicast Server
MMDS	Multichannel Multipoint Distribution Service
MID	Multiplexing Identifier Field
NI	No Increase
NNI	Network-network Interface
nrt-VBR	Non-real-time Variable Bit Rate
OAM	Operation and Maintenance
OPNET	Optimized Network Engineering Tool
PCR	Peak Cell Rate
PDU	Protocol Data Unit
PT	Payload Type
QoS	Quality of Service
RLM	Receiver-Based Layered Multicast
RM	Resource Management
RSVP	Resource ReSerVation Protocol
rt_VBR	Real-time Variable Bit Rate
SAR	Segmentation and Reassembly
SDU	Service Data Unit
SEAM	Scalable and Efficient ATM Multicast
SMART	Shared Many-to-Many ATM Reservation

SONET	Synchronous Optical Network
UBR	Unspecified Bit Rate
UNI	User-network Interface
VBR	Variable Bit Rate
VC	Virtual Channel
VCC	Virtual Channel Connection
VCI	Virtual Channel Identifier
VP	Virtual Path
VPC	Virtual Path Connection
VPI	Virtual Path Identifier
VTC	Video Teleconferencing

ACKNOWLEDGEMENTS

I would like to thank LCDR Robert Parker for his guidance and insight in producing this work. I also appreciate the use of his OPNET code for the video model, which was extremely useful.

I am eternally grateful to my dear wife and wonderful children for their love, patience, and support throughout this endeavor. I am truly blessed by them. "Who can find a virtuous woman? for her price is far above rubies." – *Proverbs 31:10* "Children are an heritage of the Lord . . . Happy is the man that hath his quiver full of them." – *127th Psalm*

Most importantly, I wish to express my humble gratitude to my Heavenly Father and Lord and Savior, Jesus Christ, for blessing me with my family, this opportunity, and the talents and abilities necessary to accomplish this work.

I. INTRODUCTION

The phenomenal growth of information technology in recent years has resulted in fundamental changes to the naval warfare paradigm. Prior to the widespread availability of high-speed digital data networks, naval war fighting was platform-based, and warfare strategies employed the use of superior forces to overwhelm and erode an adversary's forces to the point of acquiescence. Today, information technology has grown to such a degree that it can significantly leverage the outcome of a potential conflict. Network-centric warfare utilizes state-of-the-art commercial off the shelf technology (COTS) to transform individual platforms into a network of sensors and shooters, thus resulting in a shift in focus from the platform-centric to the network-centric. This paradigm shift allows forces to develop speed of command and self-synchronization and enables them to "lock-out" an adversary by rapidly achieving battle-space dominance through information superiority. Thus, an adversary with potentially superior forces can be out maneuvered and overcome through superior utilization of information technology. [1]

A. BACKGROUND

This section introduces the Navy's Information Technology for the 21st Century (IT-21) Intranet and discusses the role that Asynchronous Transfer Mode (ATM) technology plays in the IT-21 Intranet. The role of video teleconferencing (VTC) is also discussed, including the multipoint-to-multipoint multicast requirements of VTC applications. Finally, the inability of ATM to provide point-to-multipoint multicast support to VTC applications is discussed, and a solution to that problem is presented.

1. Information Technology for the 21st Century Intranet

The cornerstone of the Navy's implementation of network-centric warfare is the IT-21 Intranet. The IT-21 Intranet will provide speed of command by linking sensors, shooters, and command nodes via a reliable information network. The IT-21 Intranet will consist of ashore and afloat Local Area Networks (LANs) connected via a world wide

tactical network. The ashore and afloat LANs will be comprised of ATM backbones with minimum transmission rates of 100 Mbps and will provide ATM-to-desktop connectivity when available. Multimedia, data, and text information will be provided in near real time to the user on demand, which will facilitate speed and effectiveness of command. [2, 3]

The connectivity between the ashore and afloat LANs will be provided via a wireless interface. The wireless link between individual units of a battlegroup will likely be made through ship-to-ship line of sight (LOS) communications while connectivity with ashore LANs will likely be via a satellite interface. These methods of extremely high frequency (EHF) radio communications provide considerably less bandwidth and error robustness than that available from fiber-based LANs. Therefore the limitations provided by the wireless interface will, to a large degree, define the constraints for multimedia data transfer between individual units of a battlegroup. [4]

2. Video Teleconferencing Over ATM Networks

A powerful subset of the IT-21 Intranet multimedia capabilities will be the ability of platforms within the network to VTC with one another. VTC will enhance force efficiency, effectiveness, and reliability. Due to the interactive nature of VTC, personnel will enjoy the benefits of face-to-face communication while obviating the need to be physically co-located. Examples where VTC will provide significant advantages include collaborative tactical planning afloat, tele-medicine, remote technical assistance, and distance learning. VTC would allow such activities to occur in real time as there would no longer be the delay of hours or days previously associated with transporting the necessary personnel to and from remote locations. [3]

VTC applications operate in unicast, point-to-multipoint multicast, and multipoint-to-multipoint multicast modes. They transmit continuous multimedia data in a real-time fashion and as a result require strict bounds on both packet delay and jitter. Additionally, VTC applications usually employ compression techniques on both video and audio traffic and, therefore, require that packet losses be minimized. These characteristics of VTC applications require that the underlying network architecture

provide point-to-multipoint multicast support, Quality of Service (QoS) guarantees, and real-time support. [4]

In view of the premises that the IT-21 intranet will rely extensively on ATM backbones, that ATM services will eventually be extended to the desktop, and that VTC applications will be an integral aspect of daily operations at sea, it reasonably follows that an efficient method of employing VTC applications over ATM networks would be desirable. Although ATM does provide for point-to-multipoint communications, ATM does not currently support multipoint-to-multipoint communications per se, nor does ATM provide control signaling to support bandwidth sharing between multiple VTC sources. Consequently, a generally accepted method by which ATM may be implemented to realize multipoint-to-multipoint VTC communications has not yet been well defined. However, ATM does provide the QoS guarantees necessary to support interactive multimedia, such as VTC, as well as other types of real time variable bit rate (VBR) traffic and constant bit rate (CBR) traffic.

The real time nature and QoS guarantees of ATM, in addition to the fact that ATM will be employed from the LAN backbones to the desktop, make ATM attractive for the support of VTC applications. ATM's current inability to directly support multipoint-to-multipoint multicast requires that a "work around" be developed if the qualities of ATM are to be taken advantage of in support of robust VTC applications. This work explores the various methods available to support multipoint-to-multipoint communications over ATM, and by extension VTC, and examines the ability of one method, the Shared Many-to-Many ATM Reservation (SMART) algorithm [15,16], to provide effective multipoint-to-multipoint VTC communication with reasonable utilization.

B. THESIS OBJECTIVES

The primary objective of this thesis is to model the SMART algorithm in a simple multipoint-to-multipoint multicast tree configuration using MIL3's Optimized Network Engineering Tool (OPNET), version 5.1D. The intent is to demonstrate the ability of a

SMART multipoint-to-multipoint multicast tree, as shown in Figure I-1, to allow each user within the tree to send and receive multimedia data to and from the other users within the tree using a single virtual channel connection (VCC). SMART, as implemented by Gauthier et al. [15,16], does not specifically provide for fairness of access to the VCC by multiple sources. Therefore, a secondary objective is to examine the modifications required to the SMART algorithm so that when it is employed in a multipoint-to-multipoint multicast topology, fairness of access to the VCC for all data sources within the multicast tree will be ensured. Finally, the ability of the SMART algorithm to provide a reasonable degree of VCC utilization for the users of the multicast tree will be examined.

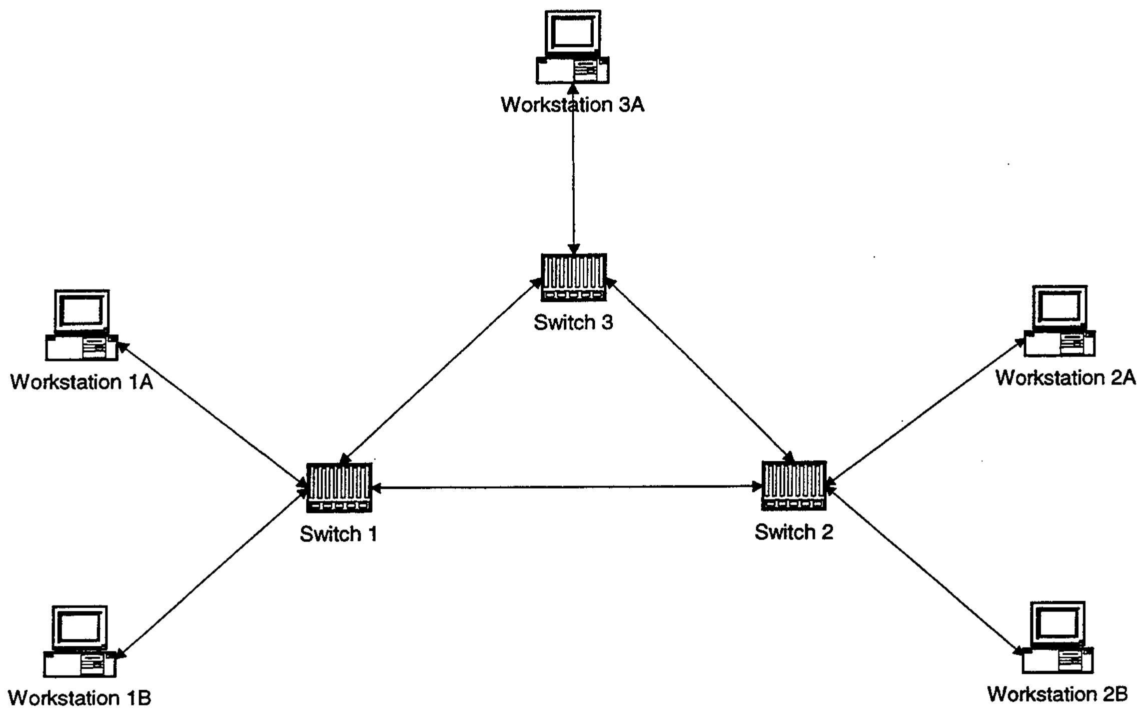

Figure I-1 SMART Multicast Tree

C. THESIS ORGANIZATION

Chapter II provides a presentation of ATM and ATM multicast. The ATM cell format, logical connections, service classes, and the ATM adaptation layer are discussed.

A brief overview of several multicast protocols is outlined. Chapter III presents the SMART algorithm in detail. SMART state variables are presented and explained. Examples are presented to explain the operation of the SMART algorithm in a simple multicast tree configuration. Chapter IV introduces modifications made to the SMART algorithm that ensure fairness for multimedia traffic in a multipoint-to-multipoint multicast tree. Chapter V provides conclusions and recommendations for future study. Appendix A provides the OPNET code for the implementation of the SMART algorithm in this work.

II. ATM AND ATM MULTICAST

Multicast is accomplished when a single source simultaneously transmits a packet to multiple receivers using a local 'transmit' operation [10]. UNI 3.1 specifies a rudimentary ATM multicast capability wherein a single source transmits information unidirectionally to multiple users in a point-to-multipoint fashion. The source is connected to the receivers via a network of ATM switches arranged as a multicast tree. The point-to-multipoint multicast tree is shown in Figure II-1.

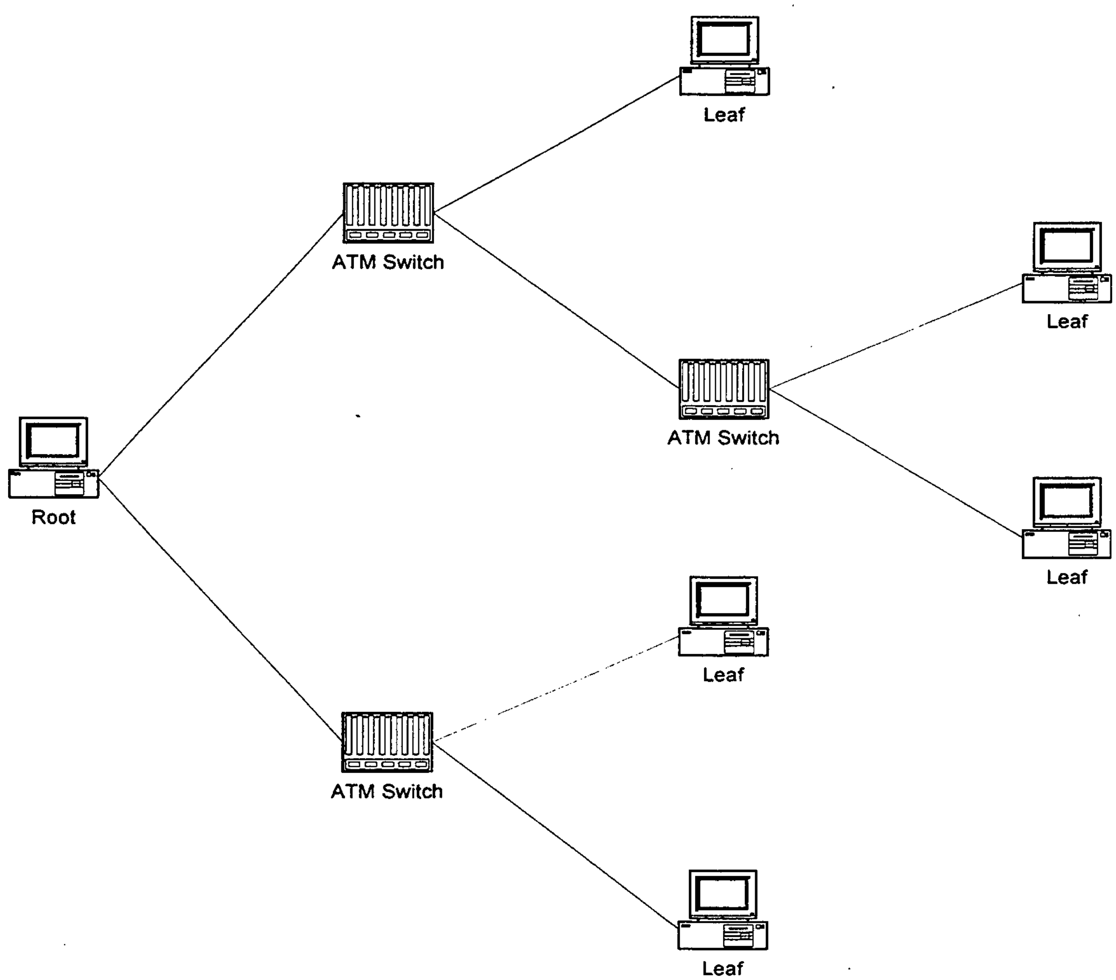

Figure II-1 ATM Point-to-Multipoint Multicast Tree

Hence, the source is referred to as the root and the receivers are referred to as leaves. The root establishes the network connection and initiates the joining of each leaf to the network. Although UNI 3.1 allows a leaf to join the tree only at the root's initiation, UNI 4.0 provides for a leaf initiated join. In either case, the root must keep track of each leaf that joins the tree. A leaf's connection may be terminated at any time by either the root or the leaf itself. [9, 11]

While ATM does provide point-to-multipoint multicast capability, neither UNI 3.1 nor UNI 4.0 provide a specification for robust multicast in a multipoint-to-multipoint sense. After briefly reviewing the ATM networking architecture, several proposed ATM multipoint-to-multipoint solutions are examined culminating with the SMART algorithm that is the focus of this thesis.

A. ATM NETWORKS

ATM is an outgrowth from the work on the broadband integrated services digital network (B-ISDN) and is similar to packet switching using X.25 and frame relay. It is a high-speed data transfer protocol designed to operate over an optical physical medium (such as the synchronous optical network, or SONET) at transmission rates which include OC-3 (155.52 Mbps) and OC-12 (622.08 Mbps). ATM is capable of providing QoS guarantees via connection oriented service. The fundamental data transfer unit is a 53-octet ATM cell. ATM cells are transmitted over virtual logical connections that are established at call setup. Many of these logical connections may be multiplexed over a single physical interface to realize a multiplexing gain. In order to provide universal utility, ATM employs a two-layered protocol architecture: the ATM layer and the ATM adaptation layer (AAL). The ATM layer is common to all services while the AAL is service dependent. [5]

1. ATM Cell Format

ATM segments data into 48-octet payloads that are appended to 5-octet control headers to produce a fixed length 53-octet cell. Evidence suggests that the relatively small fixed-length ATM cells have several advantages over larger non-fixed-length

packets including reduced queuing delay, greater switching efficiency, and easier implementation of switching mechanisms in hardware [5]. The cell control header contains the information that enables the fundamental traffic management capabilities provided by the ATM network, as well as minor error detection and correction for the control header itself. Figure II-2 shows the format for an ATM data cell exchanged at the user-network interface (UNI).

Bit Position

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Generic Flow Control</td><td colspan="4">Virtual Path Identifier</td></tr>
<tr><td colspan="4">Virtual Path Identifier</td><td colspan="4"></td></tr>
<tr><td colspan="8">Virtual Channel Identifier</td></tr>
<tr><td colspan="4"></td><td colspan="3">Payload Type</td><td>CLP</td></tr>
<tr><td colspan="8">Header Error Control</td></tr>
<tr><td colspan="8">Information Field
(48 octets)</td></tr>
</table>

Figure II-2 ATM Cell Format at the UNI [10]

The four-bit generic flow control (GFC) field is used to control cell flow at the UNI. When cells are exchanged at the network-network interface (NNI), the GFC field is replaced by expanding the virtual path identifier (VPI) field.

The eight-bit VPI field identifies a routing path within the network The 16-bit virtual channel identifier (VCI) identifies routing paths between end users.

The three bit payload type (PT) field is used to identify the type of ATM cell, as well as to provide some indication of congestion within the network. Three types of ATM cells may be passed within the network: data cells, operation and maintenance (OAM) cells, and resource management (RM) cells. OAM cells indicate virtual path (VP) and virtual connection (VC) availability and enable performance monitoring functions at the ATM layer, as well as detection, indication, and management functions at the physical layer [7].

RM cells enhance ATM's traffic management capabilities beyond those provided by the cell header. RM cells allow the network to communicate the current state of the network to the sender and source, thus allowing the source to adjust its transmission rate as necessary. Figure II-3 shows the format for an ATM RM cell used for available bit rate (ABR) traffic.

Bit Position

7	6	5	4	3	2	1	0
5 Octet ATM Cell Header							
ID							
D	BN	CI	NI	Reserved			
ECR (Two Octets)							
CCR (Two Octets)							
MCR (Two Octets)							
Reserved (37 Octets)							
Reserved						CRC-10	
CRC-10							

Figure II-3 ATM RM Cell Format [7]

The five-octet ATM cell header of the RM cell is identical to that for the data cell. Eight bits are allocated for the protocol identifier (ID) field. The single-bit direction (D) field indicates whether the RM cell is a forward RM cell or a reverse RM cell. The single-bit backward explicit congestion notification (BN), congestion indication (CI), and no increase (NI) fields are used to indicate various degrees of network congestion. The two-octet explicit cell rate (ECR), current cell rate (CCR), and minimum cell rate (MCR) fields are used by network nodes to dynamically indicate to the source acceptable cell rate transmission parameters. The ten-bit cyclic redundancy check (CRC-10) field is used for error detection in the information field. The remaining octets of information field are reserved for future use.

2. ATM Logical Connections

Routing of ATM cells is accomplished through examination of the cell header. Each ATM switch examines the VCI and VPI fields of cells it receives to determine the appropriate output port for that cell. The use of the VPI and VCI fields allows the establishment of "virtual" paths for the cells. In this manner two end users can establish a virtual connection through a series of ATM switches. The two types of virtual paths are known as virtual channel connections (VCCs) and virtual path connections (VPCs) and correspond to the VCI and VPI fields, respectively. Both VCCs and VPCs provide for the sequential, unidirectional flow of ATM cells between end users, networks, or an end user and a network. A VCC provides a single channel for data flow, whereas a VPC contains several VCCs and can provide multiple channels for data flow. The data transmitted in VPCs and VCCs can either be information or control signals between network nodes.

3. ATM Service Classes

ATM is designed to support a wide range of audio, video, and data services within a single network [8]. Due to the diverse characteristics and QoS requirements of the various types of services to be supported by ATM, the ATM forum has designated five ATM service classes. These service classes are shown in Table II-1.

Interactivity	Service Class
Real-time service	Constant bit rate (CBR)
	Real-time variable bit rate (rt-VBR)
Non-real-time service	Non-real-time variable bit rate (nrt-VBR)
	Available bit rate (ABR)
	Unspecified bit rate (UBR)

Table II-1 ATM Service Classes [9]

Real-time service is employed primarily by interactive applications, such as videoconferencing and telephony that are sensitive to both delay and delay jitter. Real-time service applications can be both smooth (CBR) and bursty (VBR) in nature. CBR

applications include broadcast quality (uncompressed) videoconferencing and interactive audio while rt-VBR applications include compressed video and audio [4].

Non-real-time service is required by applications that are not sensitive to delay and are primarily bursty in nature. Nrt-VBR applications include those that require timely responses, as with bank transactions or airline reservations. UBR uses bandwidth "leftover" from CBR and VBR connections, and provides no cell loss or delay guarantees; as such UBR is characterized as best-effort service. ABR attempts to improve upon UBR by minimizing cell loss, but without providing delay guarantees. Under each user specifies a maximum required bandwidth and a minimum cell rate (MCR), which may be zero. [5,7]

4. ATM Adaptation Layer (AAL)

Although ATM supports a wide range of video, audio, and data applications, most of these applications do not directly "map" onto the ATM layer. It is therefore necessary for an intermediate layer, the AAL, to be provided as an interface between the application and the ATM layer. The International Telecommunication Union Telecommunication Standardization Sector (ITU-T) has defined four service classes that are based on three service requirements [4]. The relationships between the service classes, service requirements, and the type of AAL are shown in Table II-2.

<table>
<tr><th></th><th>Class A</th><th>Class B</th><th>Class C</th><th>Class D</th></tr>
<tr><td>Timing Relation Required</td><td colspan="2">Required</td><td colspan="2">Not Required</td></tr>
<tr><td>Bit Rate</td><td>Constant</td><td colspan="3">Variable</td></tr>
<tr><td>Connection Mode</td><td colspan="3">Connection Oriented</td><td>Connectionless</td></tr>
<tr><td rowspan="2">AAL Protocol</td><td>Type 1</td><td>Type 2</td><td colspan="2">Type 3/4</td></tr>
<tr><td colspan="2"></td><td>Type 5</td><td></td></tr>
</table>

Table II-2 AAL Protocol Mapping To Service Classes [5]

AAL1 is used for connection-oriented CBR traffic. The remaining AAL protocols are appropriate for VBR traffic. AAL2 is intended for variable bit rate, real-time applications, such as audio and video, which require timing synchronization

between source and destination. However, development of the protocol has been delayed, as its specification was originally withdrawn and then resubmitted [19].

AAL3 and AAL4 were originally implemented as very similar protocol specifications and then merged as a single protocol specification known as AAL3/4. AAL3/4 supports both connectionless and connection oriented variable bit rate traffic and is well suited for applications with low delay requirements. AAL3/4 provides for cell flow multiplexing and interleaving over a single VCC through the use of a multiplexing identifier (MID) field. However, use of the MID field results in increased overhead and limits the potential number of users to a maximum of 1024. [5]

AAL5 is a connection-oriented protocol that supports VBR traffic. It is similar to AAL3/4, but AAL5 is not as complex and requires less overhead. AAL5 assumes that higher layers perform connection management and that the ATM and physical layers produce minimal errors. This further reduces the overhead required for AAL5. Due to the simplicity of AAL5, it is the preferred protocol for VTC traffic. [4, 5]

B. ATM MULTICAST

This section presents several existing protocols that provide ATM multicast capabilities and discusses the primary benefits and disadvantages of each. It concludes with a presentation of the SMART algorithm and explains why SMART is preferable over the other protocols examined.

1. ATM Virtual Channel Mesh

The simplest method to employ multipoint-to-multipoint multicast over ATM is to use a virtual channel (VC) mesh. A VC mesh architecture requires that each multicast source establish a unidirectional point-to-multipoint tree as shown in Figure II-2. The overlying point-to-multipoint trees constitute a mesh of VCs, with each VC corresponding to a separate root. Thus, N sources would require N overlying VC trees to create the VC mesh. This architecture has the advantages of low latency per network node and relatively high throughput. Further, the ATM signaling system is able to ensure the optimization of branching points for each ATM switch along the VC tree for each

root [4,10]. However, using a VC for each root consumes greater resources, such as channel bandwidth and system memory, and the transition of a member to or from the network creates a burst of signal messages, which contributes to significant loading on the VC mesh [4,10,12].

2. ATM Multicast Servers

Multipoint-to-multipoint multicast over ATM may also be accomplished by using a multicast server (MCS). Each root establishes an independent virtual channel connection (VCC) with the MCS, which in turn establishes and maintains a point-to-multipoint VC tree to each of the leaves, as shown in Figure II-4. An MCS has only two VCs, one for input and one for output. This architecture requires that each AAL protocol data unit (PDU) transmitted by a root be completely reassembled at the MCS prior to retransmission. This is necessary because the common-part convergence sublayer (CPCS) PDU is the smallest unit of transfer for AAL5 and does not support cell interleaving from multiple sources. The PDUs are then queued for point-to-multipoint multicast transmission via the single output VC of the MCS. [10]

The architecture as described above can easily lead to an MCS becoming a bottleneck for cell traffic. In order to alleviate this potential condition, multiple MCSs can be employed in a multicast group. Further, additional MCSs may be used as "standbys" in the event that a primary MCS fails. Thus, the judicious use of multiple MCSs can both reduce the potential for bottlenecking and increase fault tolerance. Another advantage of the MCS architecture over the VC mesh architecture is that a transition of a member to or from the MCS multicast group only requires the relatively small amount of signaling associated with either setting up or terminating a single VC link. Perhaps the most significant advantage of the MCS architecture over that of the VC mesh architecture is that it greatly reduces the number of VCs required to establish a multipoint-to-multipoint multicast group. [10,12]

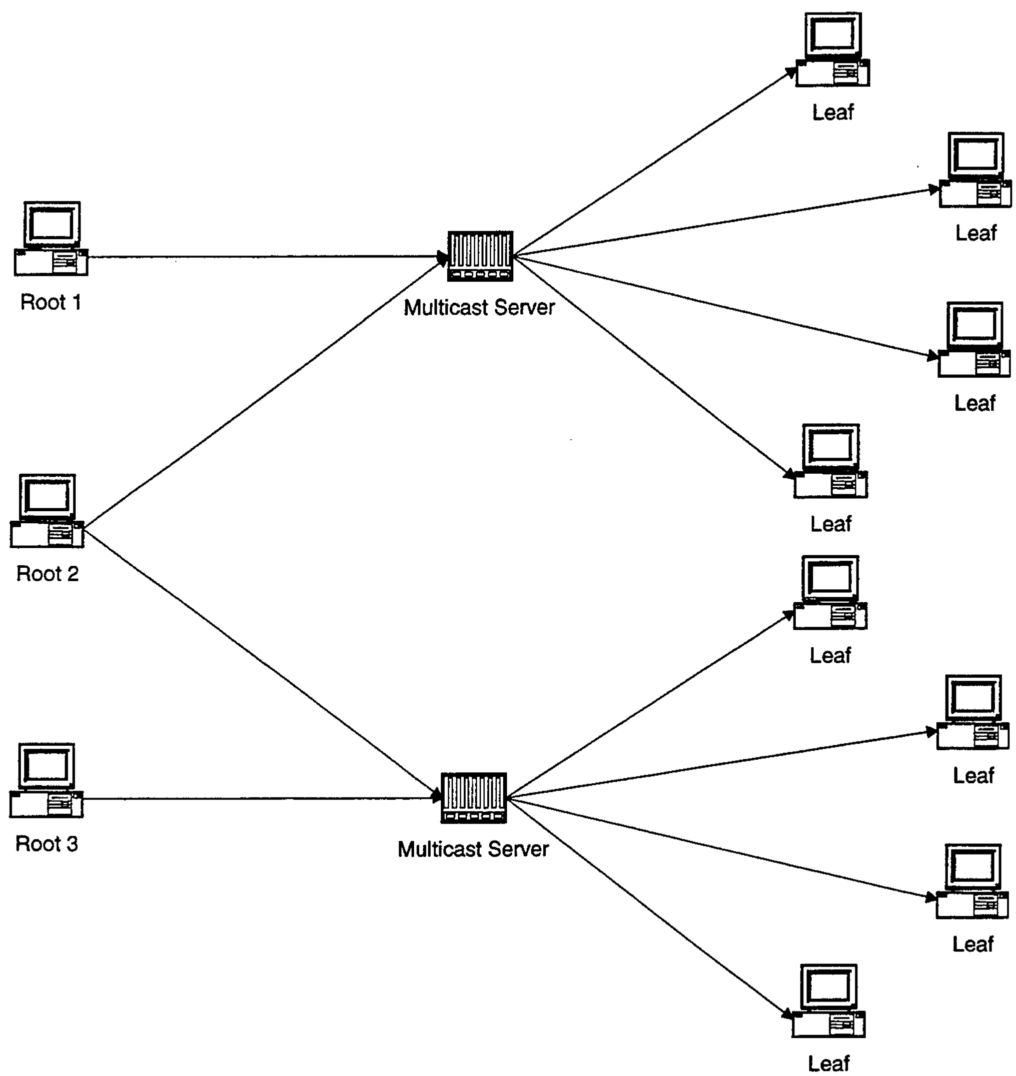

Figure II-4 ATM Multicast Using Multicast Servers

3. Scalable and Efficient ATM Multicast

Scalable and Efficient ATM Multicast (SEAM) accomplishes multipoint-to-multipoint multicast using a single "core-based" tree shared by all group members. Upon setup of the SEAM architecture, the SEAM ATM switch that is closest to the "center of mass" of the set of receivers is designated as the core. The core is the focal point for both data routing and signaling (including leaf-initiated joins to the tree) although all data and signaling do not necessarily pass through the core. SEAM uses two mechanisms, "short-

cutting" and "cut-through forwarding," to effect multipoint-to-multipoint data transfer within the tree. Short-cutting sets up the forwarding tables of the SEAM switches such that a cell received by a switch from an incoming VC link will be transmitted on all the outgoing VC links for that switch. This assumes that the switch is ready to receive cells from that particular VC link. If the switch is not ready to receive cells from that VC link then the cells are buffered until the switch is ready. Cut-through forwarding assumes the use of a block of contiguous data that is multiples of an ATM cell, hereafter known as a packet. Upon receiving the beginning cell of a packet, a SEAM switch becomes "cut-through" in that the switch is now dedicated to the VC link corresponding to the incoming packet. Cells received from any other VC links are buffered until each respective link's turn. The SEAM switch continues to forward the cells received from the cut-through VC link until the last cell of the packet is received. The last cell of the packet is indicated by the end of packet (EOP) bit (also known as the service data unit, or SDU, bit [5]) that is located in the Type AAL 5 cell's segmentation and reassembly protocol data unit (SAR-PDU). Once the last cell of a packet is received, the cut-through condition is passed in round-robin fashion to the switch's next VC link that has incoming cells pending. [12,14]

The SEAM algorithm is advantageous in that it does not require the multiple VCs of a VC mesh architecture, nor does it require the additional servers used in the MCS architecture. Additionally, SEAM is scalable and can be implemented in an ATM architecture with minor signaling modifications [12]. However issues of traffic management and reliable multicasting require further investigation and are the subject of current research [14].

4. Shared Many-to-many ATM Reservations

The Shared Many-to-Many ATM Reservations (SMART) algorithm, proposed by Gauthier et al. [15,16], provides multipoint-to-multipoint multicast communications using a single shared ATM tree and does not require additional servers or cell buffering within the network. A shared tree is setup using a single VCC to interconnect the users. SMART is capable of supporting one or many VCCs, with each VCC representing

another multicast tree. This characteristic of SMART is appealing to bandwidth shedding techniques such as those using layered video coding for VTC [4], where VCCs may be added or dropped as available bandwidth varies. SMART prevents cell interleaving without the need to reassemble the AAL Type 5 PDU or read the SAR-PDU EOP bit, and thus fully supports all ATM adaptation layers. SMART individually supports unspecified bit rate (UBR), available bit rate (ABR), constant bit rate (CBR), and variable bit rate (VBR) traffic, and respects the traffic contract in cases where QoS is specified.

ATM does not currently provide the multipoint-to-multipoint multicast capabilities required to support VTC applications. Several protocols have been presented which attempt to meet the multipoint-to-multipoint multicast need using existing ATM capabilities. Of those protocols presented, the SMART algorithm appears to be the best suited for VTC applications. A detailed description of the SMART algorithm is provided in the following chapter.

III. SMART ALGORITHM

The SMART algorithm operates completely within the ATM layer to provide multipoint-to-multipoint multicast within the ATM network. In addition to multicast capabilities, SMART provides for demand sharing between multiple sources, wherein available bandwidth is equitably distributed to the sources, although the SMART algorithm does not specify *how* this is to be accomplished. The primary function of the SMART algorithm is to control access to the shared ATM multicast tree. SMART does not ensure that an end user is ready to receive data. Additionally, SMART assumes that all signaling and routing functions required to establish and modify the multicast tree are accomplished externally to the SMART algorithm.

An in depth discussion of the SMART algorithm, originally proposed by Gauthier et al. [15,16], is presented in this chapter. This chapter essentially summarizes Gauthier's work. Modifications proposed by the author to the SMART algorithm are presented in Chapter IV.

A. OVERVIEW OF SMART

This discussion of SMART assumes the use of a single VCC corresponding to a single multicast tree, with the understanding the algorithm is scalable to multiple VCCs. SMART defines data structures in a manner similar to SEAM in that data is organized in blocks whose sizes are multiples of ATM cells. RM cells are used to delineate each block of data cells, as well as to update state information maintained within the SMART ATM switches themselves. The SMART algorithm presupposes that the end user workstations will be SMART capable and that the ATM network will be comprised of non-SMART and SMART ATM switches. The SMART algorithm is fully interoperable with non-SMART ATM switches as these switches are transparent to the SMART network architecture.

The SMART algorithm manages traffic by using RM cells to pass grants throughout the network. Before a SMART ATM workstation or a SMART ATM switch

is permitted to transmit a data block, it must hold a grant at the appropriate VC link (or port). Once that workstation or switch has transmitted the final cell of the current data block and if there is an outstanding request to transmit data from another workstation, the grant is passed (via an RM cell) to the requesting SMART workstation. Fairness issues associated with grant passing are discussed in Chapter IV.

1. The SMART ATM Switch

Each SMART ATM switch (or SMART workstation) maintains twelve bits of state information for every VC port associated with a multipoint-to-multipoint connection. This state information is used for traffic management within the multipoint-to-multipoint tree and is updated periodically by RM cells from other SMART ATM switches and SMART workstations. The state variables that are maintained for each SMART port are shown in Table III-1. Each state variable will be explained in more detail in Section III. B.

Variable	Description	Possible Values
ag	accepted grant	0,1
ar	accepted request	0,1
sg	grant to send or last sent	0,1
sr	request to send or last sent	0,1
ssn	sequence number to send or last sent	0,1,2
rg	last received grant	0,1
rr	last received request	0,1
rsn	last received sequence number	0,1,2
bias	indicates if first grant sent may be canceled	cancel, no cancel
status	indicates state of the VC link	active, inactive

Table III-1 SMART State Variables [15]

The *accepted grant* and *accepted request* state variables indicate whether that SMART port has accepted a grant or request, respectively, from the neighboring SMART

port that shares the same VC link. A value of zero indicates that a grant or request has not been accepted, whereas a value of one indicates that a grant or request has been accepted.

The *grant last sent* and *request last sent* state variables indicate whether that SMART port has sent a grant or request to transmit data, respectively, in the last RM cell transmitted to the neighboring SMART port. A value of zero indicates that a grant or request has not been sent and a value of one indicates that a grant or request has been sent.

The *last received request* and the *last received grant* state variables indicate whether that SMART port received a request to transmit data or a grant, respectively, in the RM cell last received from the neighboring SMART port. A value of zero indicates that a grant or request has not been received and a value of one indicates that a grant or request has been received.

The *sent sequence number* and *received sequence number* state variables are used to detect loss or duplication of RM cells passing grants to and from SMART ports and can take the values of zero, one, or two. The *status* state variable is used to indicate whether the VC link for a SMART port is active or not. The *bias* state variable indicates which SMART port on a VC link holds the bias and is negotiated at the set up of the SMART multicast tree.

All of the variables in Table III-1 with the exception of bias and status are originally initialized to zero. The bias variable is fixed upon establishment of the network connection and the status variable for each participating link is set to active.

State information is transmitted between SMART ports via RM cells. RM cells reflect the current values of *sg*, *sr*, and *ssn* for the SMART port from which they are sent. RM cells are transmitted periodically and in response to specific changes in certain state variables of a SMART node. Specifically, an RM cell will be immediately sent from a SMART port if any of the following events occur: the value of *sr* changes to one and *ag* is equal to zero, the value of *sg* changes to one, or the value of status changes from inactive to active and *sg* is equal to one. As explained later, transmitting RM cells

asynchronously ensures that a grant is promulgated as rapidly as possible and that data requests sent in the direction of the root will be immediately transmitted.

Upon receipt of an RM cell, a SMART port will immediately update its *rg*, *rr*, and *rsn* state variables. These state changes may in turn generate state changes in other state variables at the SMART workstation and result in the transmission of additional RM cells.

2. Dynamic Tree Initialization

As previously stated, it is assumed that the ATM multicast tree is established independent of the SMART algorithm. However, once the ATM multicast tree has been set up, it is still necessary to initialize the ports of each participating SMART workstation and SMART ATM switch in order to ensure proper cell routing. This is accomplished through dynamic tree initialization. Although each port maintains ten state variables, only six of these, *ag*, *ar*, *rsn*, *sg*, *sr*, and *ssn*, are relevant to the discussion at hand. These state variables will be represented in the format shown in Table III-2.

	grant	request	sequence number
accept:	*ag*	*ar*	*Rsn*
sent:	*sg*	*sr*	*Ssn*

Table III-2 State Variable Representation [15]

The accept row contains values for *accepted grant*, *accepted data request*, and *received sequence number* state variables and the sent row contains values for *sent grant*, *sent data request*, and *sent sequence number* state variables. In order to avoid confusion, it is important to distinguish between the *received grant* and *received request* to send data state variables and the *accepted grant* and *accepted request* to send data state variables. As will be explained in Section III. B., the *received* state variables do influence the *accepted* state variables. However it is imperative to remember that the two types of state variables are unique, and that a change in state of a *received* state variable does not necessarily immediately imply a change in state of an *accepted* state variable. Using the format presented above, a value of one is represented by the letter **G** for state variables *ag*

and *sg* and by the letter **R** for state variables *ar* and *sr*. The value of zero is represented by a minus sign for these same state variables. The *rsn* and *ssn* state variables are represented with numerical values. Shown in Figure III-1 below is an example using the above format where the SMART port has not accepted a grant but has sent one, has accepted a data request but has not sent one, its last received sequence number was one, and its last sent sequence number was two.

a:	-R1
s:	G-2

Figure III-1 State Variable Example [15]

The initialization of a simple SMART tree architecture is illustrated using the notation in Table III-2 and Figure III-1. It is initially assumed that there are no pending requests to send data, although a request to send data may be received from any workstation immediately upon completion of tree initialization. A simple multicast tree consisting of four workstations and two switches is shown in Figure III-2.

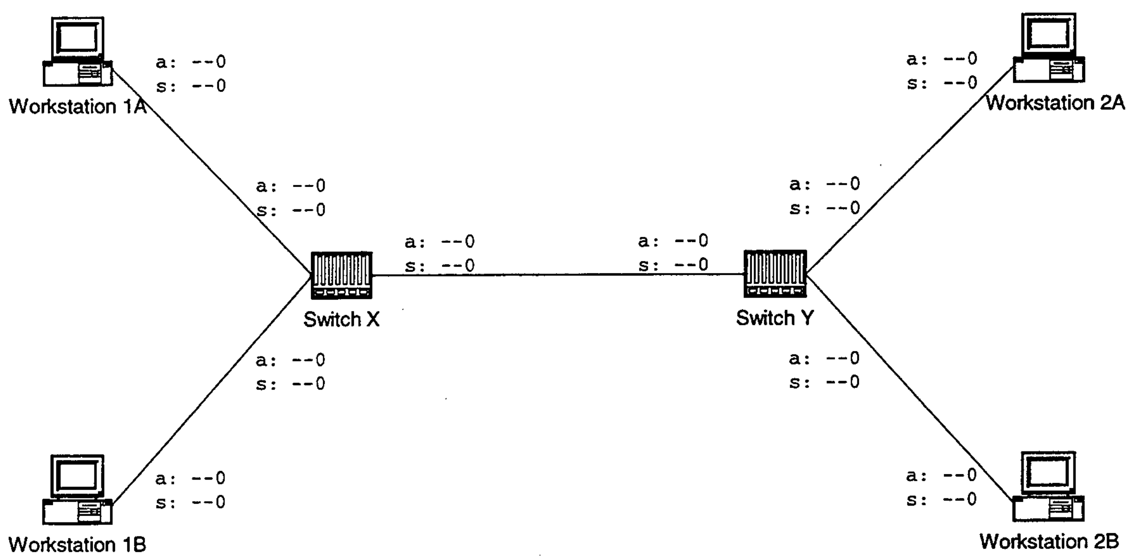

Figure III-2 SMART Multicast Tree Prior to Initialization

A single SMART port resides in each workstation and the corresponding values of each port's relevant state variables are shown adjacent to the workstation. For this

example, each SMART switch has three SMART ports, although the number of SMART ports will vary with network topology. The corresponding values of each port's relevant state variables prior to initialization are shown adjacent to the VC link attached to that port. All state variables are set to a value of zero prior to initializing the multicast tree.

Upon initialization each workstation port uses an RM cell to send a grant to its neighboring port indicating that it is ready to receive data, as shown in Figure III-3. Note that the *grant sent* state variable for each workstation port has been updated to reflect that a grant has been sent. Each workstation port that sends a grant increments its *sent sequence number* state variable by one. Each port that is ready to receive data has an arrow pointing to it along the VC link. At this point RM cells indicating the change in state variables are sent from each SMART workstation port to its respective neighbor, but the RM cells have not yet been received by the SMART switch ports.

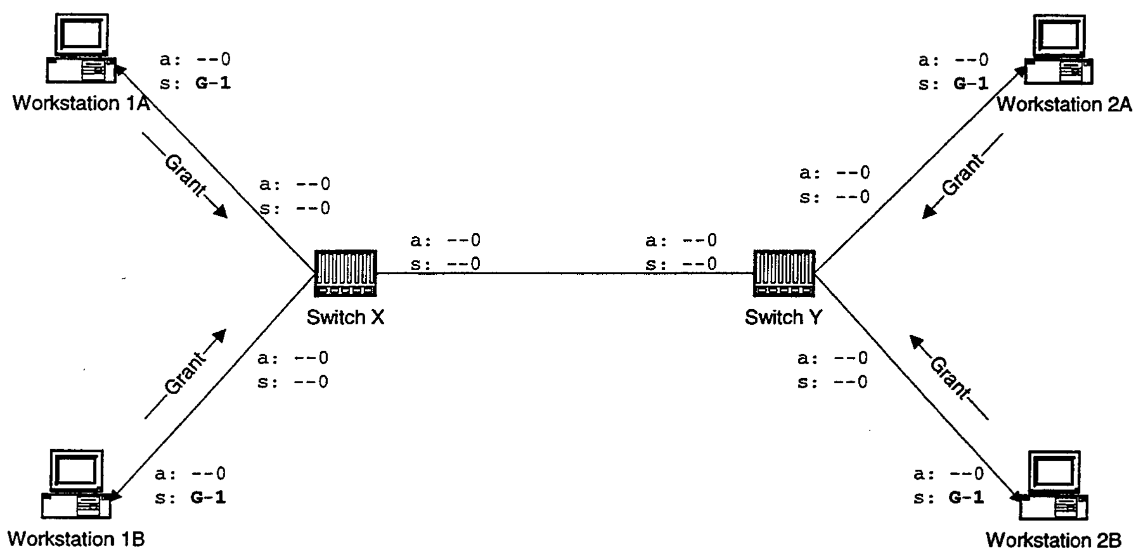

Figure III-3 Transmission of First Grant from SMART Workstations

Figure III-4 shows the state variables for each port after receipt of the grant RM cell by each SMART switch port. Each SMART switch port that has been sent a grant accepts that grant and updates its *accept grant* and *received sequence number* state variables. Each SMART switch now holds grants at two of its three ports, and as a result

meets the necessary conditions, as explained in Section III. B., to send a first grant on the remaining port.

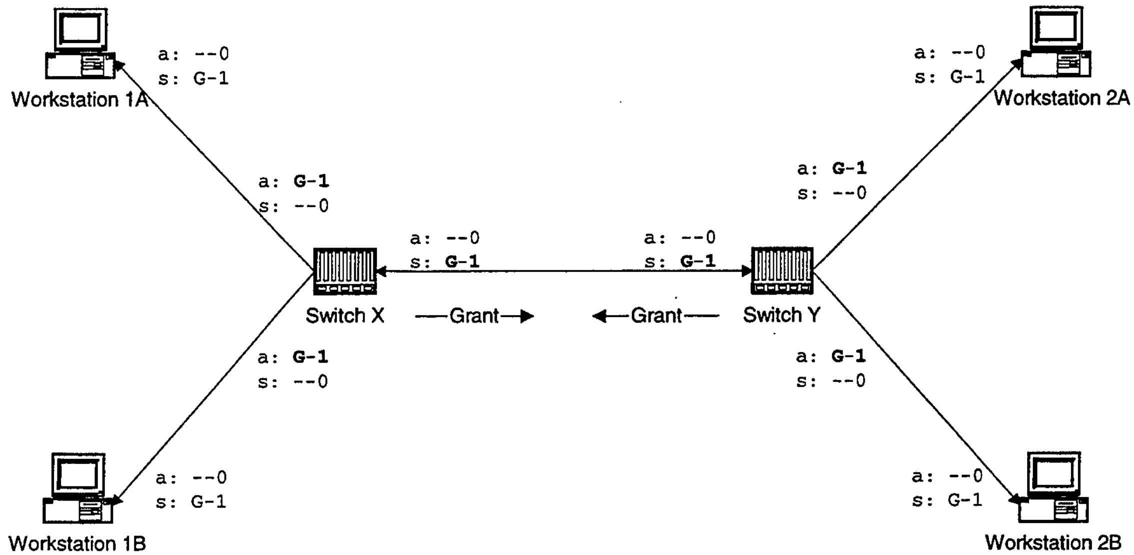

Figure III-4 Transmission of First Grant from SMART Switches

As can be seen in Figure III-4, SMART switch X and SMART switch Y have each set their *sent grant* and *sent sequence number* state variables for their respective ports on their common VC link. As a result, they have also passed grants to each other via RM cells. Now a decision must be made as to which switch actually "keeps" the grant.

At connection setup the determination is made for each VC link as to which SMART port common to the VC link will hold the bias. The bias must be set in order to determine which port on the VC link will receive and keep a grant when both ports are sending a grant to each other. Although the procedure for determining the bias is beyond the scope of this presentation, a simple method for establishing the bias must be elucidated in order to proceed with the discussion. In each case where a workstation is an end-point of a VC link, that port will not have the bias. In this instance there will be a SMART switch at the opposite end of the same VC link, and the SMART switch is always assigned the bias. Therefore, each SMART switch holds the bias on each VC link

that has a workstation on the other end. SMART switch Y is arbitrarily chosen to hold the bias on the VC link it shares with SMART switch X.

Since each workstation does not have the bias, it would not accept a first grant were one to be sent by its neighboring SMART switch. Conversely, each corresponding SMART switch node on the other end of the VC link does have the bias, and therefore would cancel a first grant it sent and accept a grant received from its respective neighboring workstation. The need for a bias on a SMART switch to SMART workstation VC link is obviated in our example since each SMART switch is initially unable to send a grant.

With respect to the VC link between both SMART switches, since SMART switch Y does have the bias, its port which shares the VC link with SMART switch X (also known as the X port of SMART switch Y) will accept the grant from SMART switch X and cancel the grant which it sent. Further the Y port of SMART switch X does not have the bias and as a result does not accept the grant from SMART switch Y.

Figure III-5 shows the nominal state of the SMART multicast tree prior to an initial request to send data. Note that the flow arrows indicate the direction in which data would flow if it were present. SMART switch Y is set up to transmit from all of its ports, while SMART switch X is set up to receive data from SMART switch Y and then relay that data to workstations 1A and 1B. All of the workstations are ready to receive data. This condition is "nominal" in that the SMART multicast tree is not ready to receive data, but it is ready to establish a multicast connection upon receipt of a request to transmit data.

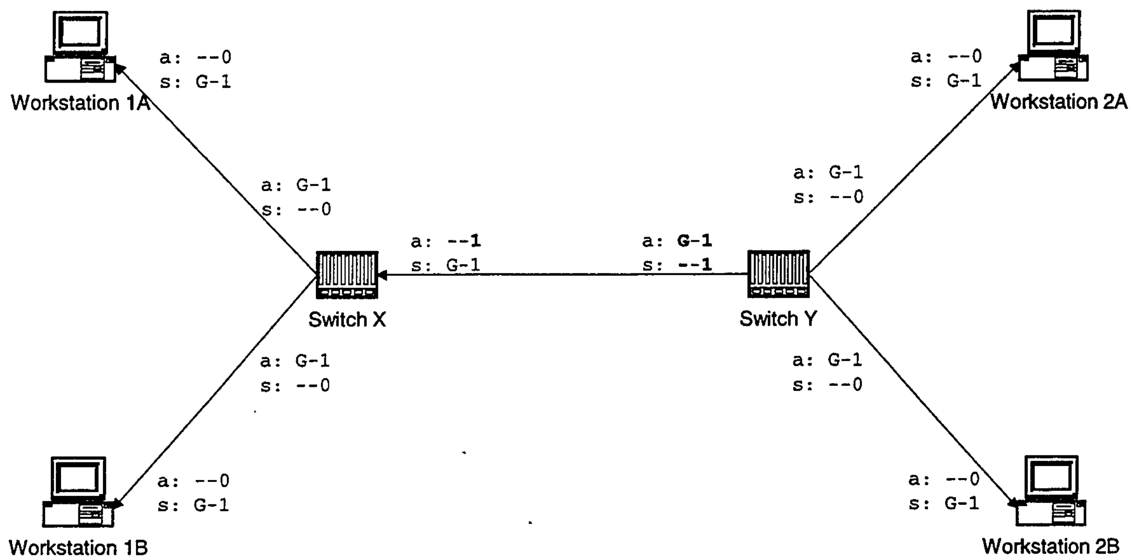

Figure III-5 Initialized SMART Multicast Tree

As shown in Figure III-6, a request to transmit data is now initiated at workstation 1A. The SMART port at workstation 1A sets its *request to send* state variable and then transmits an RM cell to SMART switch X requesting to transmit data.

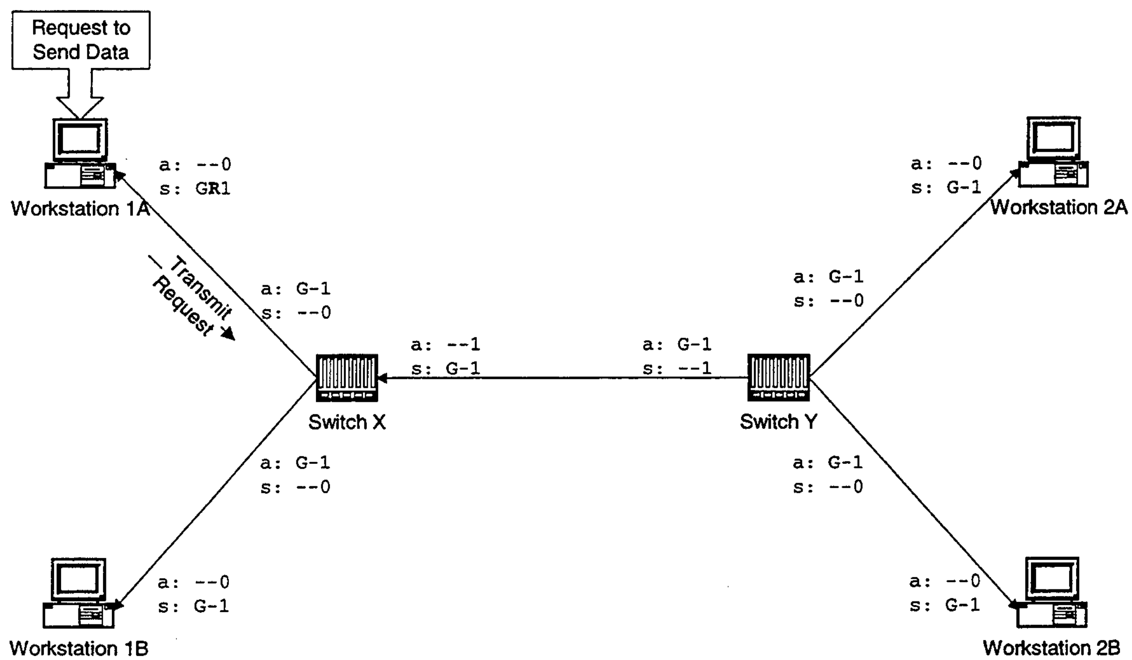

Figure III-6 Initial Request to Send Data

The request to send data is then received at SMART switch X, which updates its *received request* state variable for its 1A port. SMART switch X then generates a request to send data at its 1B port and Y port. The *request to send* state variable for each of these ports is set and RM cells are sent from SMART switch X to workstation 1B and to SMART switch Y, as shown in Figure III-7.

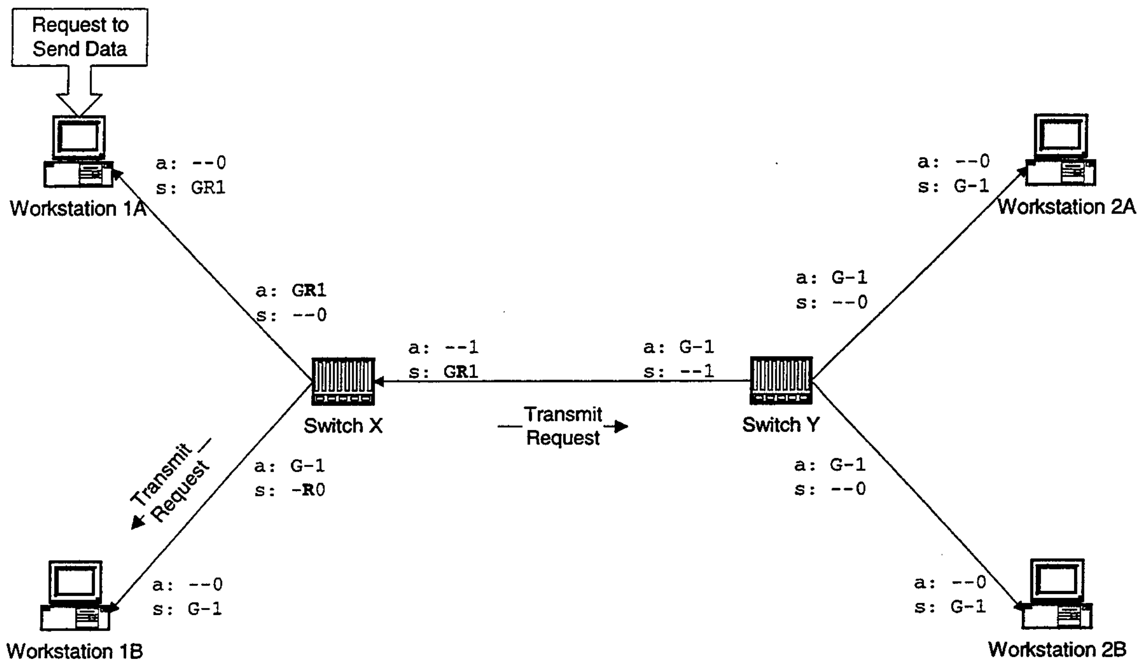

Figure III-7 Promulgation of Request to Send Data

Once the RM cells transmitted by the ports of SMART switch X have been received at their destinations, the corresponding state variables are updated, as shown in Figure III-8. Workstation 1B has accepted the request to transmit data and has set its *received request* state variable. SMART switch Y sets its *received request* state variable at the corresponding port, and then forwards this request by sending RM cells to workstations 2A and 2B. The *request to send* state variable for the corresponding ports of SMART switch Y are also set.

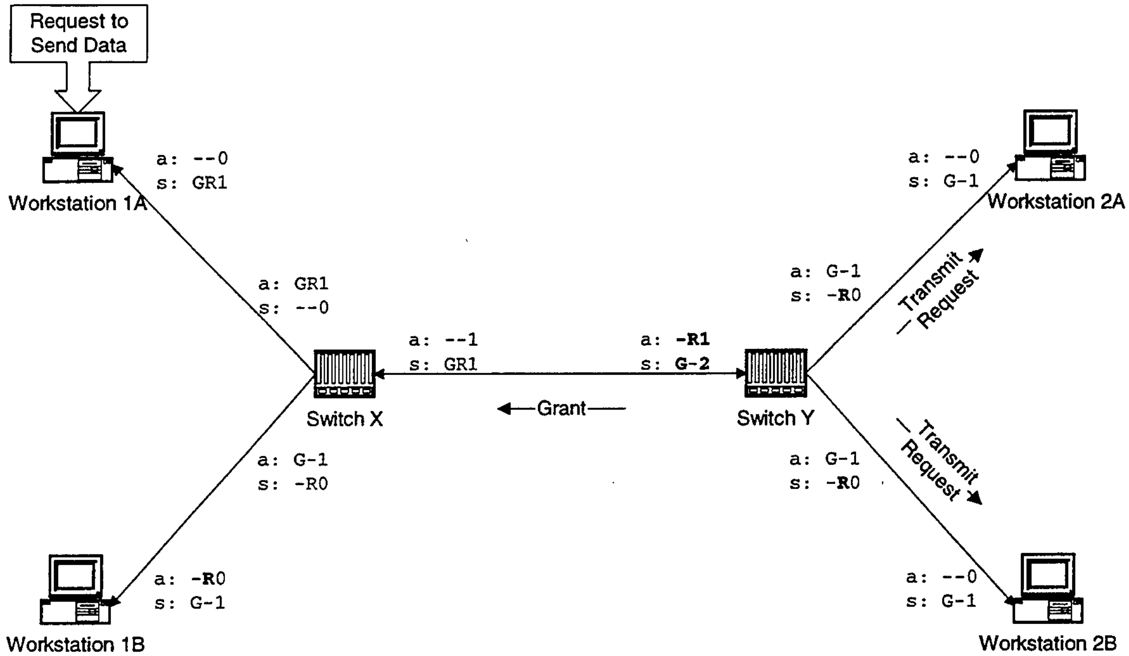

Figure III-8 Transmission of Grant From SMART Switch Y to SMART Switch X

The receipt of a data request triggers an additional event at SMART switch Y, the generation of a grant transmission. SMART switch Y already holds a grant at each of its active ports, and as explained in Section III. B., meets the conditions to transmit a grant now that a data request has been received. SMART switch Y resets the *accept grant*, sets the *send grant*, and increments the *sent sequence number* state variables at its X port. SMART switch Y sends the grant to SMART switch X via an RM cell.

Once SMART switch X receives the grant at its Y port, it resets its Y port *sent grant*, sets its *accept grant*, and increments its *received sequence number* and *sent sequence number* state variables. SMART switch X now holds a grant at each of its active ports, in addition to the data request at its 1A port. Thus, SMART switch X meets the conditions required to send a grant via its 1A port. SMART switch X then resets its *accept grant*, sets its *sent grant*, increments its *sent sequence number* state variables at its 1A port and sends a grant to workstation 1A using an RM cell, as shown in Figure III-9.

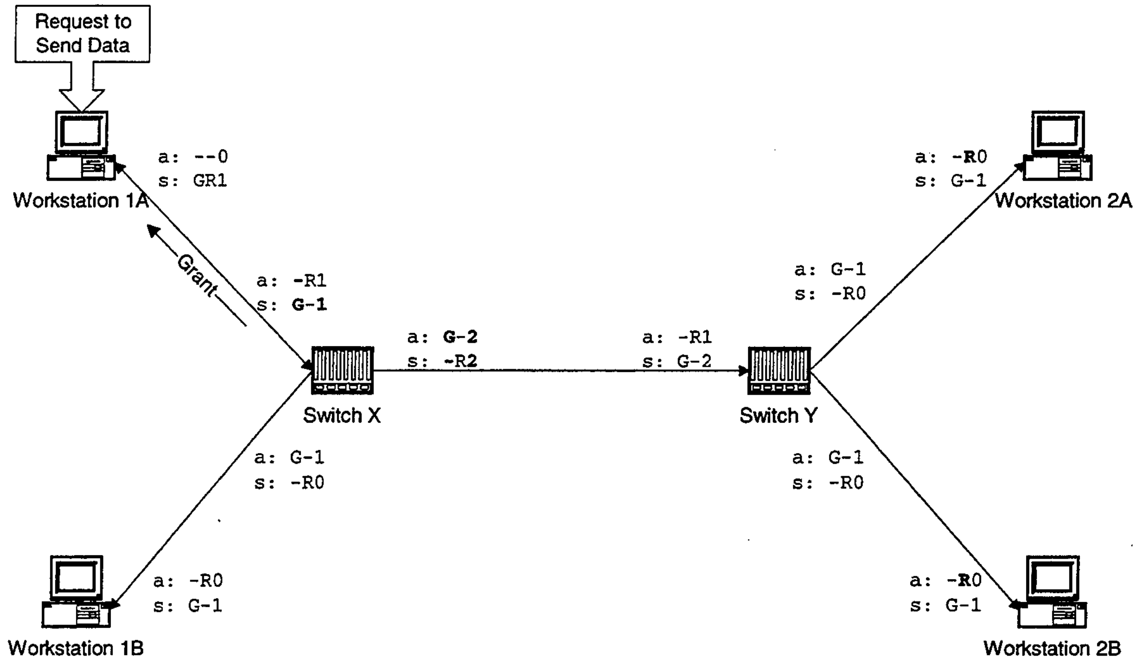

Figure III-9 Transmission of Grant from SMART Switch X to Workstation 1A

Upon receipt of the RM cell containing the grant, workstation 1A resets the *sent grant*, sets the *accept grant*, and increments its *received sequence number* and *sent sequence number* state variables at its SMART port, as shown in Figure III-10. Now that workstation 1A holds the grant it is permitted to send data via the SMART multicast tree. The flow paths in Figure III-10 indicate that the SMART multicast tree is set up to receive data from workstation 1A and transmit that data to all the other workstations in the tree, namely workstations 1B, 2A, and 2B.

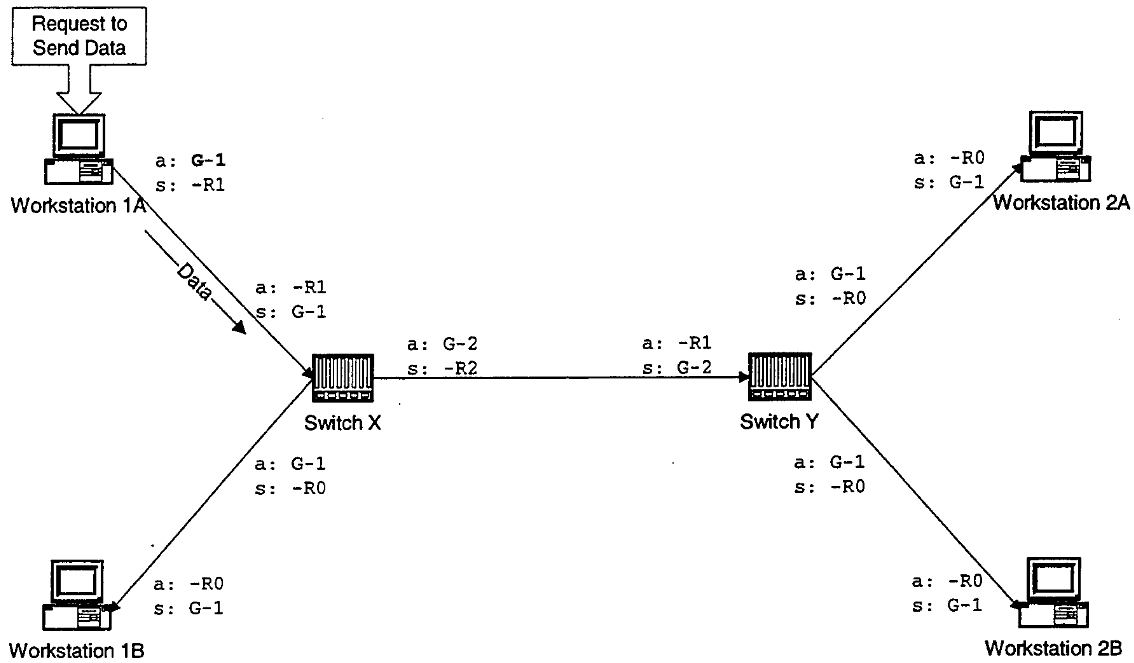

Figure III-10 Data Transmission to SMART Multicast Tree from Workstation 1A

The SMART multicast tree depicted in Figure III-10 will remain in its current state until workstation 1A no longer needs to send data or another request to transmit data is received. The latter case will now be examined and is illustrated in Figure III-11. Workstation 2A now initiates a request to send data. As before with workstation 1A, the state variables of the SMART port of workstation 2A are updated and an RM cell is sent to SMART switch Y with a request to transmit data. SMART switch Y receives the RM cell, updates its state variables (as previously described), and promulgates the request to send data as in RM cell transmitted to SMART switch X, as shown in Figure III-12. SMART switch X receives the RM cell, updates its state variables, and promulgates the request to send data (again via an RM cell) to workstation 1A, as shown in Figure III-13. Workstation 1A receives the request to send data and updates its port state variables as shown in Figure III-14. Workstation 1A continues to transmit data cells until the end of the current data block is reached.

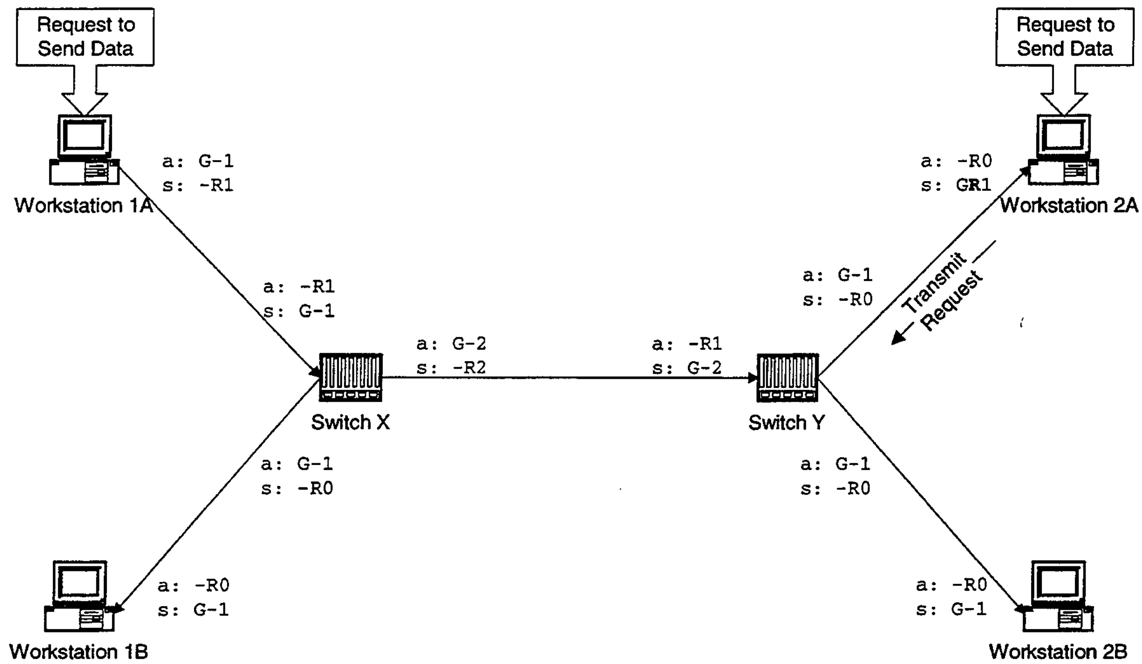

Figure III-11 Second Request to Send Data

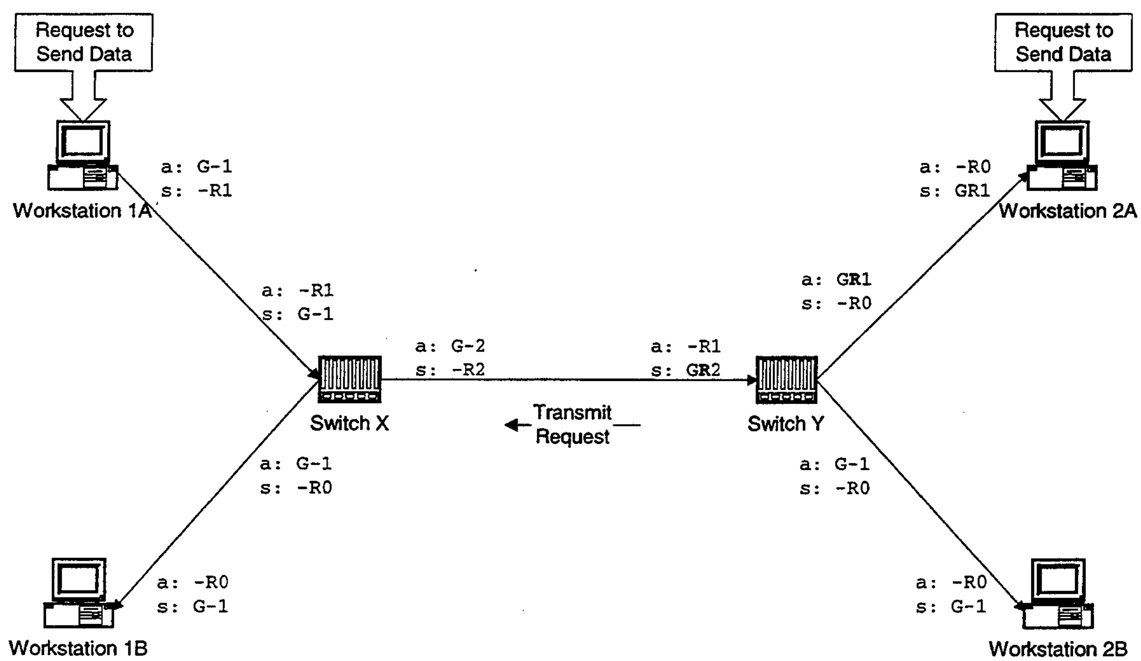

Figure III-12 Promulgation of Request to Send Data to SMART Switch X

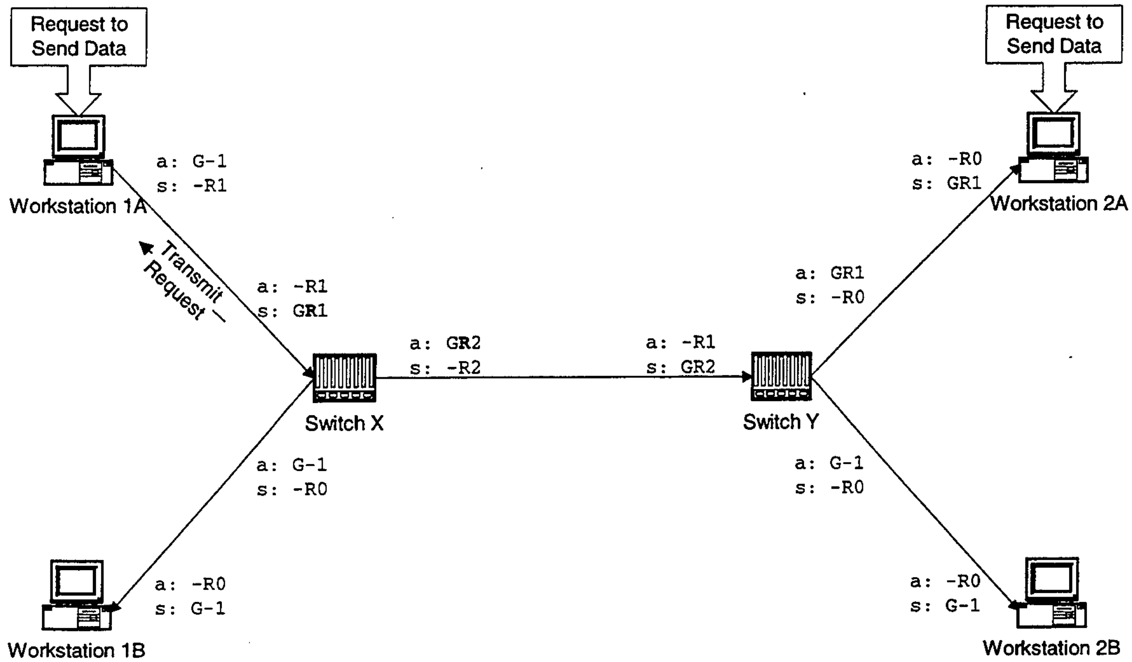

Figure III-13 Promulgation of Request to Send Data to Workstation 1A

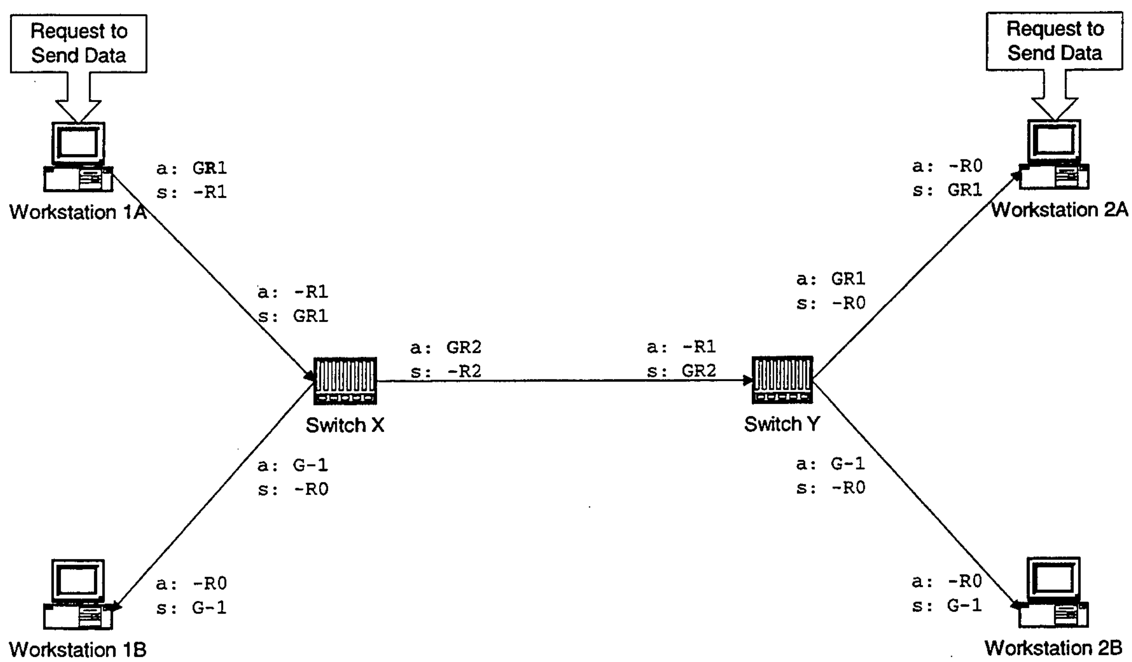

Figure III-14 Receipt by Workstation 1A of Request to Send Data

Upon completing the transmission of the last data cell of the current data block, workstation 1A recognizes the request to send data at its SMART port. Workstation 1A then resets its SMART port's *accept grant*, sets its *sent grant*, increments its *sent sequence number* state variables and then transmits the grant to SMART switch X, as shown in Figure III-15. Note that it is no longer possible to send data along the multicast tree, as workstation 1A is now ready to receive data while the rest of the SMART multicast tree is still oriented to receive data from workstation 1A.

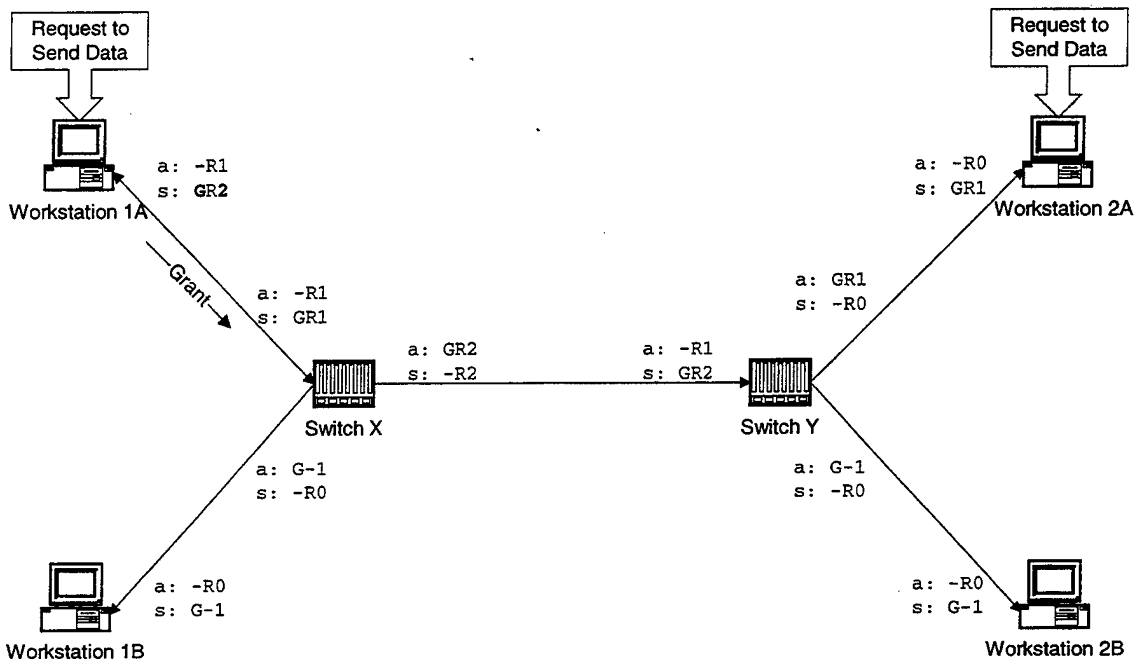

Figure III-15 Release of Grant by Workstation 1A

Upon receipt of the grant from workstation 1A, SMART switch X port 1A updates its state variables. SMART switch X now holds a grant at each of its active ports and has a request to transmit data at both its 1A port and its Y port. It is therefore necessary for the SMART switch to recognize that although workstation 1A has an outstanding request to send data, this request is not as "old" as the request by workstation 2A. This quandary is not addressed by Gauthier et al. [15,16], and is explored in more detail in Chapter IV. It is assumed that the SMART switch recognizes that the request to be honored is that of workstation 2A. SMART switch X then updates its Y port state

variables and sends a grant to SMART switch Y, as shown in Figure III-16. Again, as the grant is passed along the SMART multicast tree, the data path is reconfigured at each VC link.

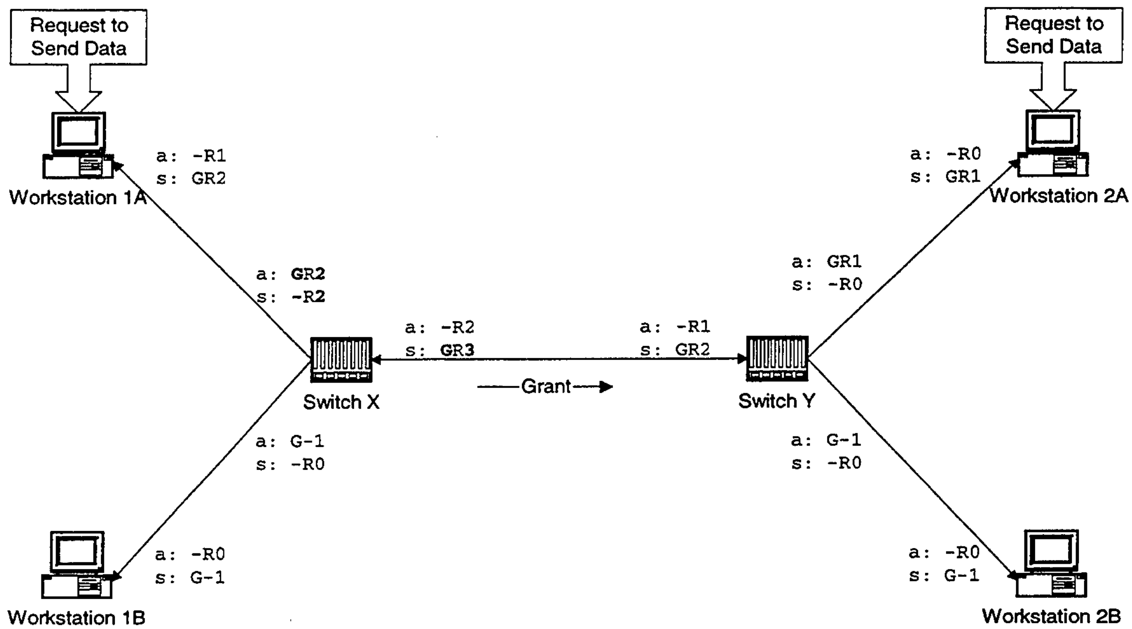

Figure III-16 Passing of Grant from SMART Switch X to SMART Switch Y

The grant is then received by SMART switch Y, which updates its port X state variables. SMART switch Y is now confronted with the same dilemma as was SMART switch X. That is that SMART switch Y now holds a grant at each port and has a request to transmit data at two ports: port X and port 2A. As a result of the fairness algorithm explained in Chapter IV, that SMART switch Y "knows" that port 2A is the correct port to send the grant through. SMART switch Y then updates its port 1A state variables and transmits the grant to workstation 2A, as shown in Figure III-17. Upon receipt of the grant, workstation 2A updates its state variables.

Figure III-18 shows that the SMART multicast tree is now configured to receive data from workstation 2A and transmit it to workstations 1A, 1B, and 2B. Workstation 2A will transmit data cells until the end of its data block. Upon transmission of the last cell in the data block, workstation 2A will pass the grant back to workstation 1A in reverse order of the manner just described.

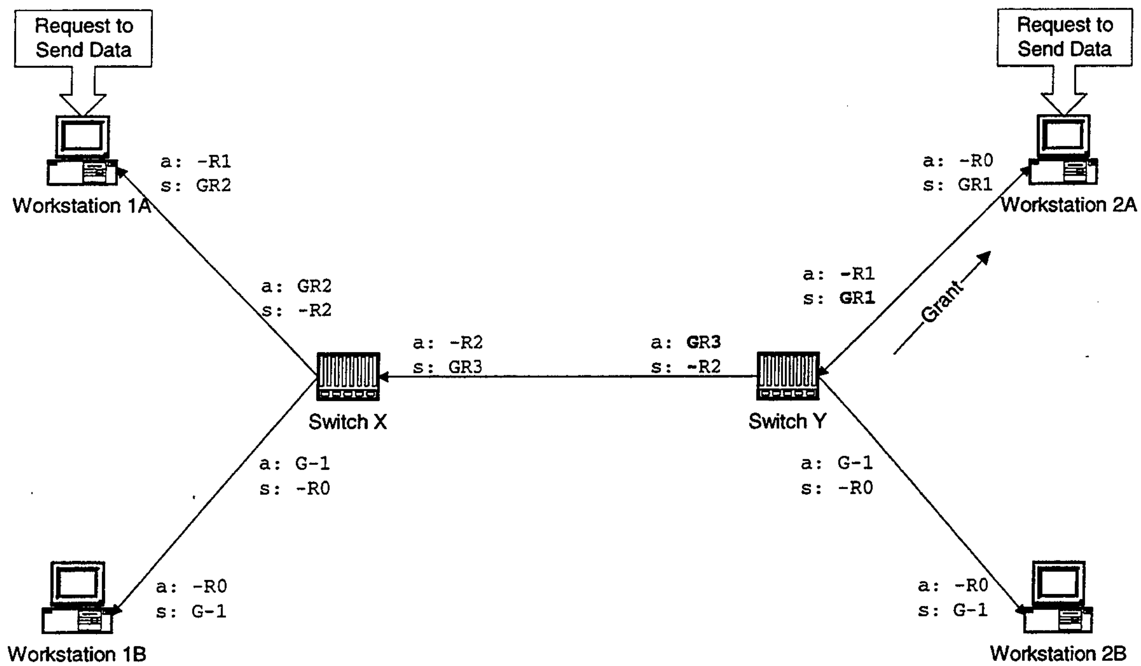

Figure III-17 Transmission of Grant from SMART Switch Y to Workstation 2A

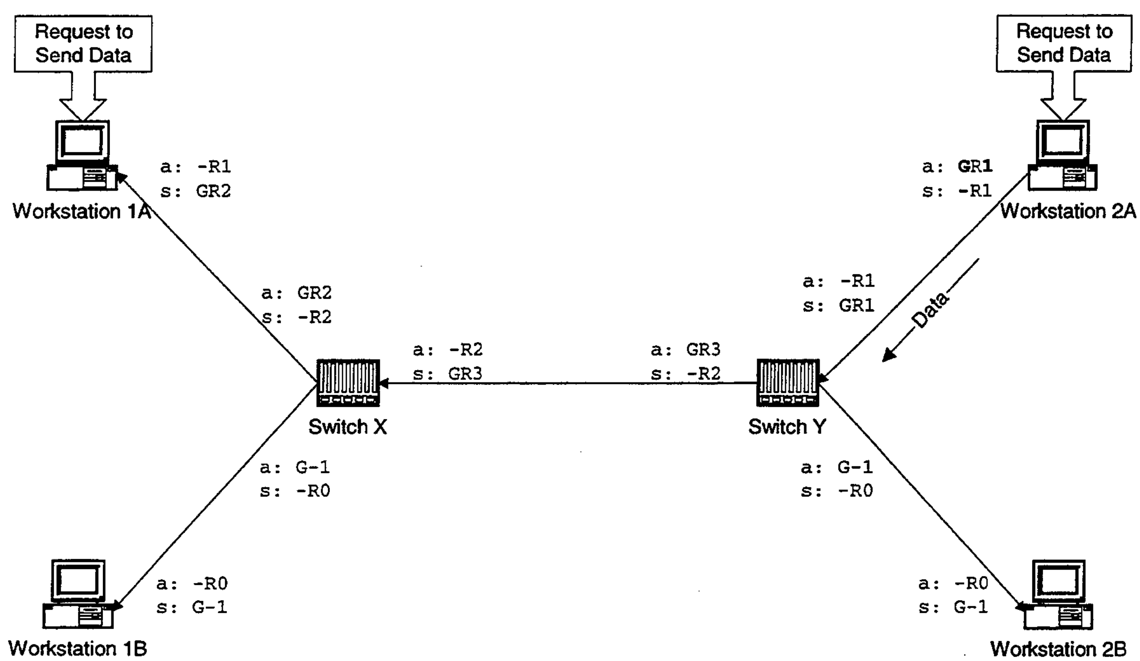

Figure III-18 Data Transmission to Multicast Tree from Workstation 2A

As the previous discussion demonstrates, the SMART algorithm provides a method whereby a single VCC may be used to provide multipoint-to-multipoint multicast services to an ATM tree. Simply put, this method is based on passing a grant between any of the workstations that need to send data. However, the algorithm as defined by Gauthier et al. [15,16], does not address fairly sharing the grant when more than one workstation needs to transmit data simultaneously. Fairness issues are explored in Chapter IV.

B. SPECIFICATION OF SMART STATE VARIABLES

The state variables for each SMART port are manipulated by several finite state machines associated with that port. These finite state machines take their input from the state variables of the associated SMART switch or SMART workstation port. The input state variables are updated when an active SMART port receives an RM cell directing a change in one or more of the port's state variables. The resultant change can result in a finite state machine transition, which in turn results in the manipulation of one or more output state variables, and in some cases the transmission of an RM cell. The affected state variables may or may not be from the same port that was initially updated by receipt of an RM cell.

This section describes the operation of the SMART finite state machines. For notational purposes, a double equal sign, = =, will represent the testing of a variable, while a single equal sign will indicate the assignment of a value to a variable.

1. The Use of Sequence Numbers

Each port maintains a sequence number that is local to the VC link associated with that port. The sequence number records the total number of grants (modulo N) sent and received by the SMART port. The purpose of the sequence number is to allow the SMART port to handle RM cell loss and duplication.

Consider the following example as shown in Figure III-19. SMART switches X and Y send each other a first grant in conjunction with initialization of the SMART multicast tree. The RM cell sent from SMART switch X is lost while SMART switch X

receives the RM cell sent from SMART switch Y. No transition occurs at SMART switch X upon receipt of the grant, as SMART switch X does not hold the bias. SMART switch X then sends a periodic RM cell, which contains the same information as the previously lost RM cell, to SMART switch Y. SMART switch Y then accepts the grant from SMART switch X (since Y holds the bias) and cancels its first grant.

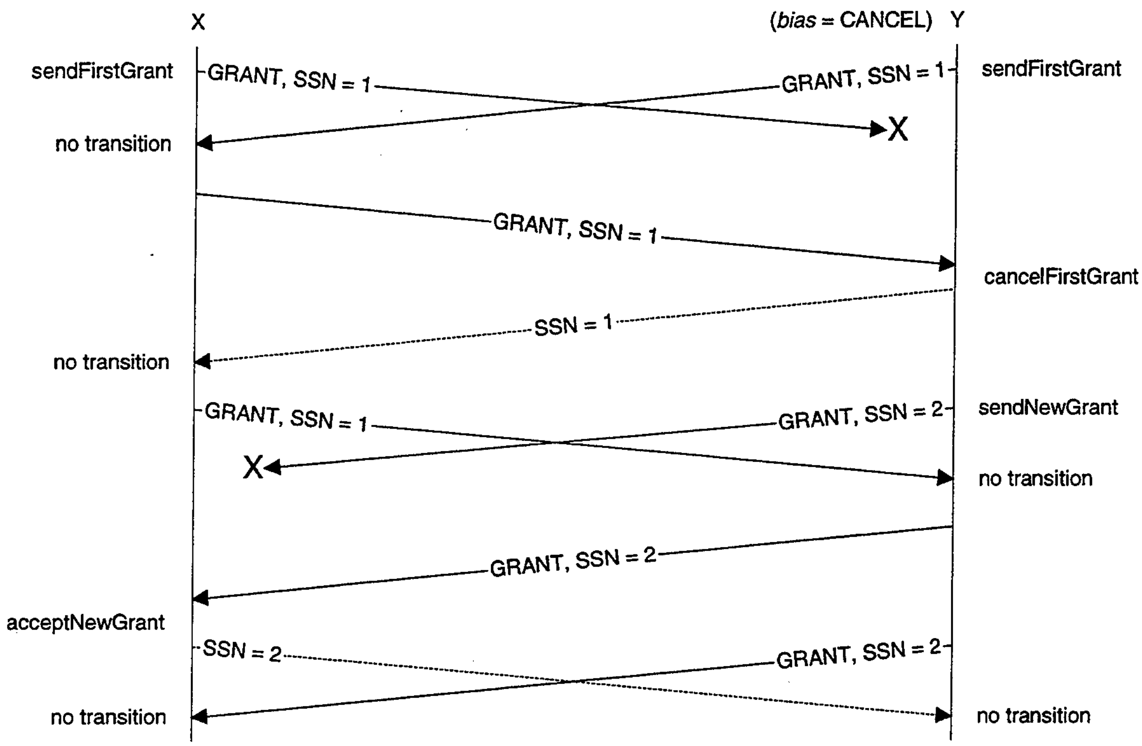

Figure III-19 Sequence Numbering and Cell Duplication/Loss [15]

SMART switch Y then sends an RM cell, that is subsequently lost, transferring the grant to SMART switch X. At the same time SMART switch Y attempts to transfer the grant to SMART Switch X, SMART switch X sends a periodic RM cell to SMART switch Y. SMART switch Y ignores the grant from SMART switch X, since the received sequence number is out of order. SMART switch Y then sends a periodic RM cell containing the same information as the lost cell to SMART switch X. SMART switch X accepts the grant from SMART switch Y since this received sequence number is in order. SMART switches X and Y then each simultaneously send periodic RM cells, neither of which result in a state transition when received.

The dashed lines in Figure III-19 represent RM cells that do not need to be sent. The crosses represent lost RM cells.

Sequence numbering must be at least modulo 3 to prevent ambiguity when transmitting grants via RM cells. Figure III-20 illustrates the reason for this. SMART switch X sends a new grant to SMART switch Y, which is accepted. SMART switch Y then sends a new grant to SMART switch X, which is accepted. Finally SMART switch X sends a grant to SMART switch Y, which is accepted. If the sequence numbering were modulo 2, the sent sequence number for the last grant sent by SMART switch X would be zero, which is the same as the sent sequence number for the previous grant sent by SMART switch X. In this case, SMART switch Y would not be able to distinguish between a new grant sent from SMART switch X or a periodic RM cell reflecting the conditions of the earlier grant transmission. Therefore, the grant sequence numbering must be at least modulo 3.

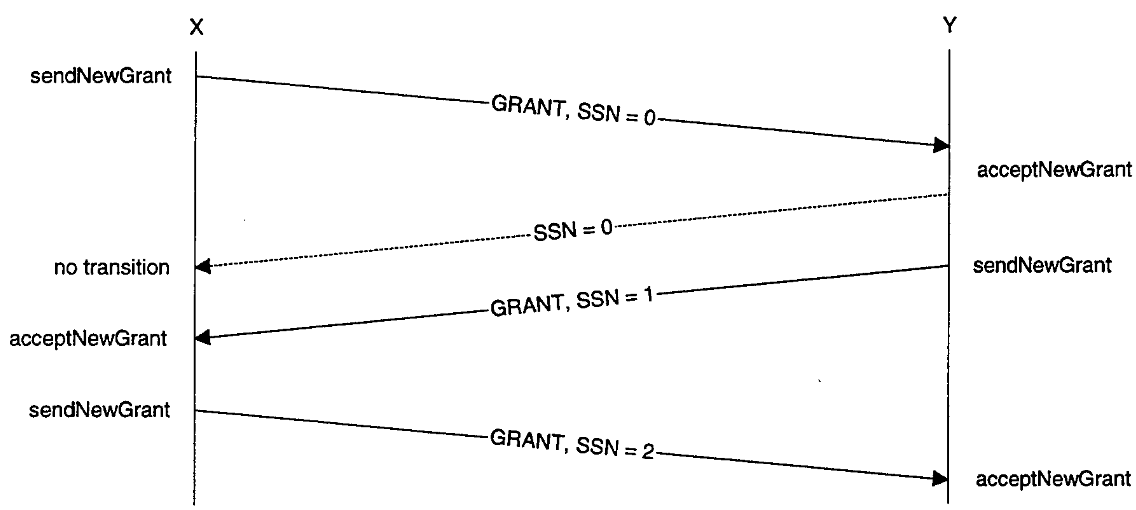

Figure III-20 Modulo 3 Sequence Numbering [15]

2. When to Send a First Grant

As discussed earlier, a grant is first sent from several SMART ports as the SMART multicast tree is being initialized. The transition **sendFirstGrant** occurs at a port if and only if that port has not previously sent a grant ($sg_n == 0$), the port is active

($status_n$ = = 0), and all other active ports associated with that SMART switch have accepted a grant (ag_m = = 1 for all $m \neq n$ such that $status_m$ = active). A SMART workstation has only one port and will automatically send its first grant if the link is active.

Once the **sendFirstGrant** transition has occurred, the port *sent sequence number* state variable is incremented ($ssn_n = ssn_n + 1 \; mod \; 3$), the state variable for that port is set ($sg_n = 1$), and an RM cell is immediately transmitted from that port. The required conditions and subsequent actions for the transition are summarized in Table III-3, where m and n are the port numbers for the switch.

conditions	$status_n$ = = active $sg_n = = 0$ $ag_n = = 0$ ag_m = = 1 for all $m \neq n$ such that $status_m$ = = active
actions	$sg_n = 1$ $ssn_n = ssn_n + 1 \; mod \; 3$

Table III-3 SendFirstGrant Transition [15]

3. How to Cancel a First Grant

An active SMART port n that has sent a first grant (sg_n = = 1) may cancel that grant upon receipt of a grant from the neighboring port on the same VC link (rg_n = = 1) if and only if two additional conditions are met. First, the port canceling the grant must hold the bias ($bias_n$ = = cancel) for that VC link. Second, the received sequence number must equal the sent sequence number ($rsn_n = = ssn_n$). Once these conditions have been met, the **cancelFirstGrant** transition will clear the SMART port's *sent grant* state variable ($sg_n = 0$) and set its *accept grant* state variable ($ag_n = 1$). These conditions and actions are summarized in Table III-4.

Conditions	$status_n == \text{active}$ $sg_n == 1$ $rg_n == 1$ $rsn_n == ssn_n$ $bias_n == \text{cancel}$
actions	$sg_n = 0$ $ag_n = 1$

Table III-4 CancelFirstGrant Transition [15]

4. When to Send a New Request

A SMART port n that does not currently hold an active request to transmit data ($sr_n == 0$) may send a new request to transmit data only under the following conditions. In the case of a SMART workstation, the request may be initiated ($sr_n = 1$) once the workstation is ready to transmit data. In the case of a SMART switch, the request may be initiated if any other port m of that switch has received a request to transmit data ($\sum_{m \neq n} ar_m > 0$). The conditions and actions associated with the transition are summarized in Table III-5.

An RM cell containing the request is sent immediately if the port does not hold a grant ($sg_n = 0$). Otherwise an RM cell is sent at the next periodic sending opportunity.

conditions	$sr_n == 0$ $\sum_{m \neq n} ar_m > 0$
actions	$sr_n = 1$

Table III-5 SendNewRequest Transition [15]

5. How to Accept a New Request

An **acceptNewRequest** transition occurs if and only if the SMART port n has received a request to transmit data from another port ($rr_n == 1$) and the SMART port does not already hold a previous request ($ar_n == 0$). Once the request to send data has

been accepted ($ar_n = 1$), other ports of the same SMART switch may immediately be eligible to activate the **sendNewRequest** transition. Table III-6 summarizes the conditions and actions for the acceptNewRequest Transition.

conditions	$rr_n == 1$ $ar_n == 0$
actions	$ar_n = 1$

Table III-6 AcceptNewRequest Transition [15]

6. When to Send a New Grant

An active SMART port *n* that has not recently sent a grant ($sg_n == 0$) and has received a request to transmit data ($ar_n == 1$) will activate the **sendNewGrant** transition if and only if all the active ports *m* associated with that SMART switch have received a grant ($ag_m == 1$ for all *m* such that $status_m ==$ active). An active SMART workstation that has not recently sent a grant and has received a request to transmit data will activate the **sendNewGrant** transition once it has completed transmitting the current data block. Upon activation of the **sendNewGrant** transition the SMART port will set its *sent grant* state variable ($sg_n = 1$), clear its *accept grant* state variable ($ag_n = 0$), increment its *sent sequence number* state variable ($ssn_n = ssn_n + 1 \ mod\ 3$), and immediately transmit an RM cell to the neighboring SMART port. The **sendNewGrant** transition is summarized in Table III-7.

conditions	$status_n ==$ active $sg_n == 0$ $ar_n == 1$ $ag_m == 1$ for all m such that $status_m ==$ active
actions	$sg_n = 1$ $ag_n = 0$ $ssn_n = ssn_n + 1 \ mod\ 3$

Table III-7 SendNewGrant Transition [15]

7. How to Accept a Grant

An active SMART n port will accept a grant if and only if it has received a grant ($rg_n = = 1$) from its neighboring SMART port and if the received sequence number from the neighboring SMART port is one increment higher than its own sent sequence number ($rsn_n = ssn_n + 1 \ mod \ 3$). Acceptance of the grant activates the **acceptNewGrant** transition which sets the SMART port's *accept grant* state variable ($ag_n = 1$), resets its *sent grant* state variable ($sg_n = 0$), sets its *accept request* state variable equal to its *received request* state variable ($ar_n = rr_n$), and sets its *sent sequence number* state variable equal to its *received sequence number* state variable ($ssn_n = rsn_n$). The conditions and actions for the **acceptNewGrant** transition are summarized in Table III-8.

conditions	$status_n = =$ active $rg_n = = 1$ $rsn_n = ssn_n + 1 \text{ mod } 3$
actions	$ag_n = 1$ $sg_n = 0$ $ar_n = rr_n$ $ssn_n = rsn_n$

Table III-8 AcceptNewGrant Transition [15]

8. When to Cancel a Request

The **cancelRequest** transition occurs if and only if the SMART port n has sent a request to transmit data ($sr_n = = 1$) and the other ports m for that SMART switch do not hold requests to transmit data ($\sum_{m \neq n} ar_m = = 0$). In the case of a SMART workstation the transition occurs if and only if its port has sent a request to transmit data and the workstation no longer has data to send. The **cancelRequest** transition resets the *sent request* state variable for that port ($sr_n = 0$).). The conditions and actions for the **cancelRequest** transition are summarized in Table III-9.

Conditions	$sr_n == 1$ $\sum_{m \neq n} ar_m == 0$
Actions	$sr_n = 0$

Table III-9 CancelRequest Transition [15]

The SMART algorithm provides a method whereby the bandwidth of a single VCC may be shared between multiple SMART workstations over a simple multipoint-to-multipoint multicast tree. This is accomplished through the operation of various state variables at each SMART port and through the exchange of state information (via RM cells) between SMART ports that are common to a VC link. Each state variable provides input a state machine, which in turn manipulates one or more different state variables.

Although the SMART algorithm has been shown to provide multipoint-to-multipoint multicast over ATM, it does not provide for fair access to the VCC. Additionally, the SMART algorithm requires modification before it can be extended to more complex multicast trees. These issues are addressed and resolved in Chapter IV.

IV. A "SMARTER" ALGORITHM

The intent of this work is to implement the previously described SMART algorithm in a simulation environment while demonstrating that access to a single VCC by the participating workstations can be fairly distributed. In order to better display the robustness of the SMART algorithm, a slightly more complex SMART multicast tree architecture is presented here than that of [15,16]. Another SMART switch is added to the previous architecture to create a delta configuration of SMART switches as shown in Figure IV-1. Although each SMART switch may serve one or more workstations, the new SMART switch serves only one workstation. The delta configuration SMART multicast tree is advantageous over the previous configuration in that this configuration is more readily extended to SMART multicast tree configurations of greater complexity.

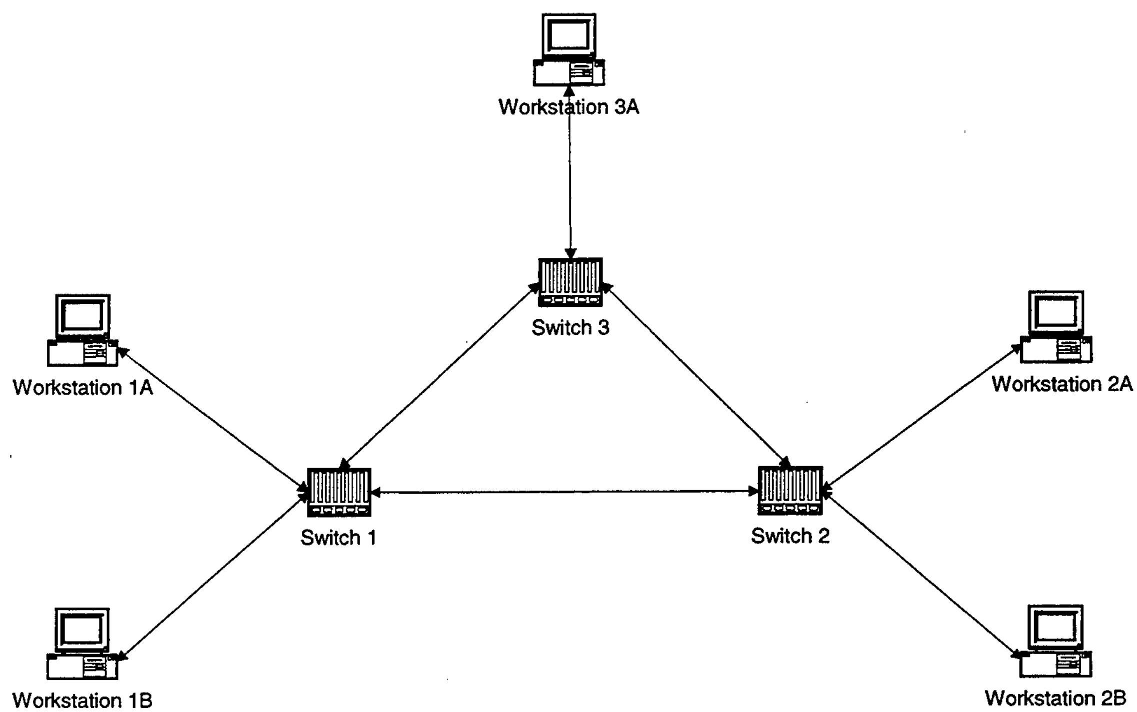

Figure IV-1 Delta Configuration SMART Multicast Tree

The delta configuration implies that each SMART switch must be capable of interacting with more than one SMART switch and more than one workstation. This

configuration is more demanding of the SMART algorithm than the previous configuration, where each SMART switch was only required to interact with one neighboring smart switch. The requirements of this new configuration, combined with the need to ensure fairness in passing the grant between workstations, necessitated the addition of new capabilities to the SMART algorithm.

A. MODIFICATIONS TO ORIGINAL SMART ALGORITHM

Three primary concerns were identified while addressing the problem of implementing the SMART algorithm in such a way as to support VTC over a delta configuration SMART ATM network. These concerns were cell delay, fairness, and cell routing.

1. Cell Delay

Excessive cell delay during a VTC can reduce the interactive quality of the audio and video information to unacceptable levels. As a result, a limit must be set with respect to end-to-end cell delay. A one-way end-to-end cell delay of up to150 ms has been declared by ITU-T G.114 to be "acceptable for most user applications" [17]. Therefore, an end-to-end cell delay of 150 ms was chosen as an upper limit for data cells for this work. The principal delay of concern for this work is cell queuing delay, and this was the only delay accounted for. Other contributors to overall delay, such as transmission delay, processing delay, propagation delay, etc, were not taken into account in the interest of simplicity. The effects of these delays would, of course, yield results less optimistic results than those obtained in this work.

Queuing delay results when a SMART workstation is generating data cells for transmission but does not hold the grant. The data cells are therefore queued until the workstation receives the grant to transmit. If the workstation is prevented from receiving the grant for an excessive period of time, then the data cells at the head of the queue tend to exceed the maximum permissible end-to-end delay. Once this point is reached, it is useless to transmit the data cell, as it would unnecessarily occupy bandwidth and arrive too late to be of use. Therefore, any data cells in the SMART workstation queue that

exceed the maximum permissible delay are deleted from the queue before transmission. To accomplish this, each SMART workstation maintains a cell tracker that monitors the oldest data cell in the queue. When it is determined that a queued data cell will exceed the maximum permissible delay, that data cell is removed from the queue and deleted, at which point the timer will begin tracking the next oldest data cell.

2. Fairness

A key concern for multipoint-to-multipoint multicast over a single VCC is the issue of fairness. Since only one SMART workstation can hold the grant at any one time, it is important that possession of the grant be rotated among the workstations in such a manner as to minimize total cell loss from all workstations. A simple approach wherein the grant is passed in "round-robin" token fashion provides some utility but does not ensure minimal cell loss, especially in cases where each workstation has varying transmission rates. Further, in cases where the workstations may or may not be transmitting, time is wasted passing the grant to a workstation that has no data to send.

A second approach would be to pass the grant only to workstations that have a request to transmit data pending. Each SMART switch would be required to "remember" which of its ports with outstanding requests to transmit data had received a data transmission least recently, and when the grant became available, it would be next passed to this port. This is more effective than the first approach, but it favors workstations that transmit at lower data rates, as their cell loss ratio will be less than that of workstations transmitting at higher data rates

A third approach would be to combine the second method with some measure of a workstation's queue size. The grant could then be passed based on a combined metric of a workstation's queue size and the last time that the workstation transmitted. However, this approach does not reflect the actual age of data cells within the queue.

The final approach, which was the approach implemented for this model, is to pass the grant based on a metric comprised of a queue size component and a cell queue age component. Since each workstation queue already tracks the oldest data cell in the queue, this statistic is readily available. The queue size statistic is also easily obtainable.

The *queue metric* was heuristically developed using various queue age window sizes and weighting methods. The optimal method uses 15 age windows, each 10 ms in width. Each age window corresponds to the data block length, which will be explained later. As the oldest data cell in the queue becomes older, it is assigned a larger age weight. The age weight increases exponentially with each successive age window, as shown in Table IV-1. The age weight is then multiplied by the queue size to obtain the *queue metric* state variable, *qm*. This state variable is used to determine the next workstation to which the grant will be passed.

Cell Age	Weight
0 ms ≤ Age ≤ 10 ms	1
10 ms < Age ≤ 20 ms	3
20 ms < Age ≤ 30 ms	7
30 ms < Age ≤ 40 ms	20
40 ms < Age ≤ 50 ms	55
50 ms < Age ≤ 60 ms	148
60 ms < Age ≤ 70 ms	403
70 ms < Age ≤ 80 ms	1097
80 ms < Age ≤ 90 ms	2981
90 ms < Age ≤ 100 ms	8103
100 ms < Age ≤ 110 ms	22026
110 ms < Age ≤ 120 ms	59874
120 ms < Age ≤ 130 ms	162754
130 ms < Age ≤ 140 ms	442413
140 ms < Age ≤ 150 ms	1202604

Table IV-1 Queue Cell Age Weights

Each SMART workstation passes its current *queue metric* in every RM cell that it transmits. When a SMART switch receives an RM cell, it updates the stored value for

the *queue metric* corresponding to that port. Each SMART switch must also forward the largest *queue metric* for the workstations it serves to its neighboring SMART switches. This information is passed via every RM cell sent by one SMART switch to another. Thus each SMART switch maintains a table of *queue metrics* for each SMART workstation that it serves, as well as its neighboring SMART switches. Once a SMART switch meets the conditions to pass a grant, it must examine the table of *queue metrics* it holds for the ports which have outstanding requests to transmit data. If only one request is held, then the grant is passed to that port. If more than one request to transmit data is held, then the grant is passed to the port with the largest *queue metric*. The required conditions and actions for the modified **sendNewGrant** transition are illustrated in Table IV-2.

conditions	$status_n == \text{active}$ $sg_n == 0$ $ar_n == 1$ $ag_m == 1$ for all m such that $status_m == \text{active}$ $qm_n > qm_m$ for all m such that $ar_m == 1$
actions	$sg_n = 1$ $ag_n = 0$ $ssn_n = ssn_n + 1 \; mod \; 3$

Table IV-2 Modified SendNewGrant Transition

A second aspect of fairness is the data block size. If the data block size is too large, then other workstations that have data to transmit may be unfairly forced to drop data cells while another source holds the grant. If the block size is too small, then an inordinate amount of time will be spent passing the grant, which would result in less efficient use of available bandwidth.

The data block size is determined by the amount of time that a SMART workstation is allowed to hold the grant. If two or more SMART workstations have data to transmit over the VCC, then the SMART workstation holding the grant must relinquish

the grant after a set period of time so that other SMART workstations may transmit their data. The minimum amount of time that a SMART workstation may hold the grant is known as the grant hold time. Once a SMART workstation holding the grant exceeds its grant hold time or its queue is empty of data cells, it determines whether it has received any requests to transmit data. If a request to transmit data exists, then the SMART workstation will pass the grant upon completion of transmission of the current data cell. If no request to transmit data is present, the SMART workstation will continue to hold the grant until a request to transmit data is received, at which time it will pass the grant (after it has completed transmission of the current data cell).

Initially, a heuristic approach indicated that a grant hold time of 10 ms was optimal for multimedia data in the scenarios examined. Further investigation revealed that the ideal grant hold time is not a constant, but rather a variable dependent upon the average queue size of a single source and the capacity of the VCC when operating at maximum utilization. In order to determine the optimum grant hold time, a single workstation is allowed to transmit while the others remain silent. VCC bandwidth is then manipulated to the point to where the maximum possible utilization is obtained for the cell loss desired. The average queue size for the workstation is then observed. The average queue size divided by the optimum VCC bandwidth yields the optimum grant hold time. This grant hold time results in the optimal data block size for any number of active sources on the multipoint-to-multipoint multicast tree.

The optimum method for determining data block size was not discovered until the majority of the results for this work had already been determined using a 10 ms grant hold time. Therefore, the results presented in this work are for a grant hold time of 10 ms. The improvement in VCC utilization ranges up to about two percent using the optimal method for determining data block size and therefore should be a subject of future investigation.

3. Cell Routing

The SMART multicast delta configuration presents difficulties not encountered by the configuration presented in [15,16]. The primary difficulty associated with the

delta configuration is the unnecessary replication and retransmission of redundant cells. By way of illustration, consider the case where a workstation sends a request to transmit data. Since the request to transmit data is sent via an RM cell, the network configuration for data cell flow is irrelevant. Additionally, since state variable operation has previously been discussed in detail, this aspect of the discussion is assumed to be understood.

As shown in Figure IV-2, workstation 1A promulgates a request to transmit data to the network. This request is received and replicated by SMART switch 1, which then transmits the request to workstation 1B and SMART switches 2 and 3. SMART switches 2 and 3 then promulgate the request to each other and to their respective workstations, as shown in Figure IV-3. SMART switches 2 and 3 then retransmit the request again, passing it to their respective workstations, each other and SMART switch one. This situation quickly results in an endless cascade of RM cell transmissions as the request transits the delta of SMART switches.

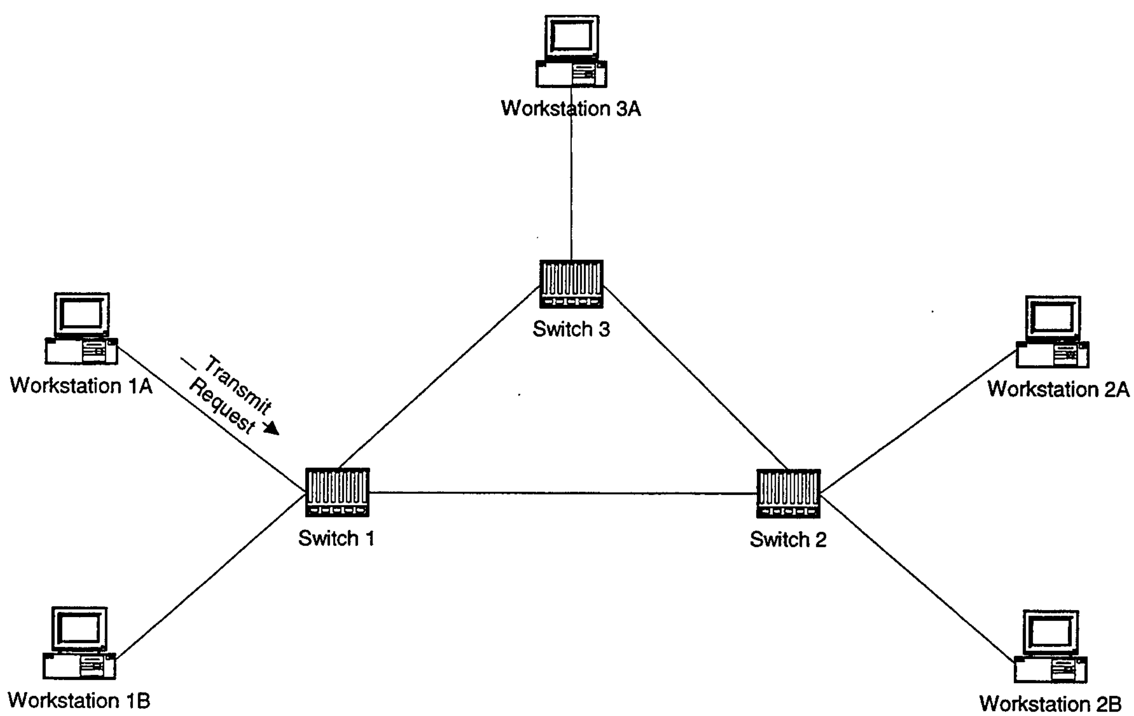

Figure IV-2 Request to Transmit Data

The delta configuration creates a topology wherein a request to transmit data from a single source can result in a perpetual regeneration of that RM cell as it continues around the delta of SMART switches. As similar situation would occur with data cells were a method not implemented to overcome this shortcoming.

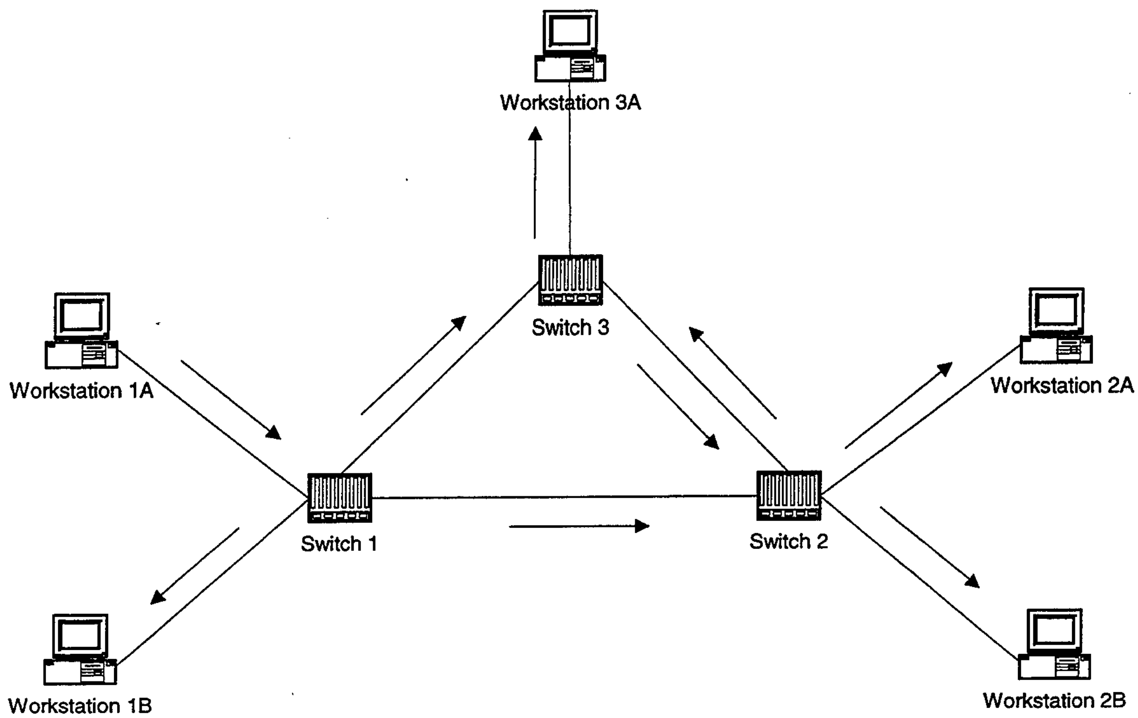

Figure IV-3 Promulgation of Request to Transmit Data through Network

In order to prevent the above scenario, a method was developed to selectively and dynamically deactivate VC links for data cell transmission. This changes the multicast tree topology in a dynamic nature. This method is specific to the delta configuration and is explained below. It is assumed that more complex networks will have previously established routing tables that are maintained by each SMART switch.

Each RM cell generated by a workstation is required to contain a field identifying the SMART switch that serves that workstation. For example, workstation 1A would be identified with a one, workstation 2B with a two, and so on. The purpose of this identifier is to facilitate the SMART switches in reconfiguring the SMART multicast tree. The intent is to inactivate the VC link opposite the SMART switch which first

receives the data request, thus preventing the unnecessary perpetual duplication of RM cells.

In order to describe the reconfiguring method, it is assumed that the SMART multicast tree has been initialized but that no requests to transmit data have been transmitted. A request to transmit data is initiated at workstation 1A and passed to SMART switch 1. This is then promulgated to workstation 1B and SMART switches 2 and 3. SMART switches 2 and 3 then promulgate the request to their respective workstations. SMART switches 2 and 3 also recognize that the request to transmit data originated from a workstation served by SMART switch 1, and as a result inactivate the VC link common to SMART switches 2 and 3. This is shown in Figure IV-4 with the inactivated link indicated by a dashed line.

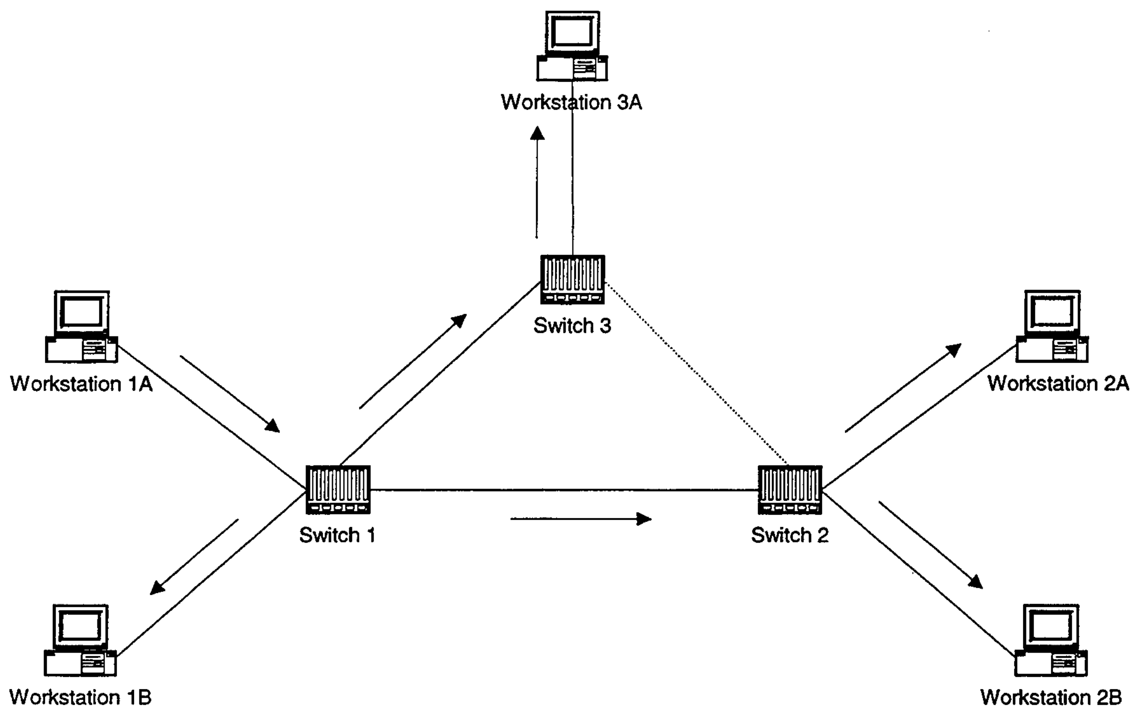

Figure IV-4 Request to Transmit Data With Link Inactivated

The grant is promulgated to workstation 1A as previously described, using the most direct route (as determined by pre-established routing tables) from the SMART switch holding grants at each of its ports. In other words, if the grant is received from

either SMART switch 2 or 3, it will be sent via SMART switch 1 to workstation 1A. Once the grant is received by workstation 1A, it will transmit data with the SMART multicast tree configured as shown in Figure IV-5.

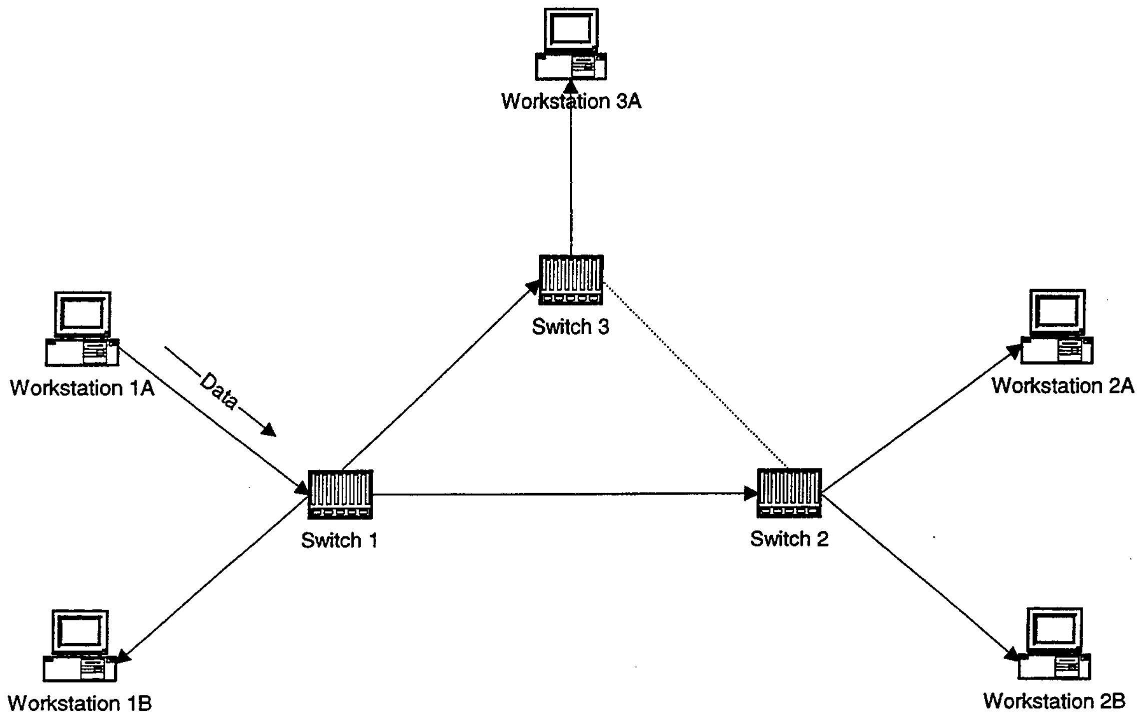

Figure IV-5 Data Transmission from Workstation 1A

This configuration still allows requests to transmit data to be promulgated, although not necessarily by the most direct route. For example, while workstation 1A would receive a request to transmit data from workstation 2B via the most direct route, workstation 3A would not. This is inconsequential, as the only workstation that "needs" to receive the request is the workstation that holds the grant. The SMART multicast tree will remain in this configuration until workstation 1A relinquishes the grant.

It will now be assumed that workstation 3A has promulgated a request to transmit data. Once workstation 1A has finished transmitting the last data cell of its data block, it passes the grant to workstation 3A. Although workstation 3A would be able to transmit data using the configuration of Figure IV-5, this would not be the most effective method. Therefore, the SMART multicast tree must be reconfigured. Once the grant is received

by SMART switch 3, the VC link between SMART switches 2 and 3 is reactivated, restoring the nominal configuration to the network. Once the grant is received by workstation 3A, the workstation transmits RM cells to SMART switches 1 and 2, inactivating the VC link between them. The network is now configured to receive data from workstation 3A, as shown in Figure IV-6.

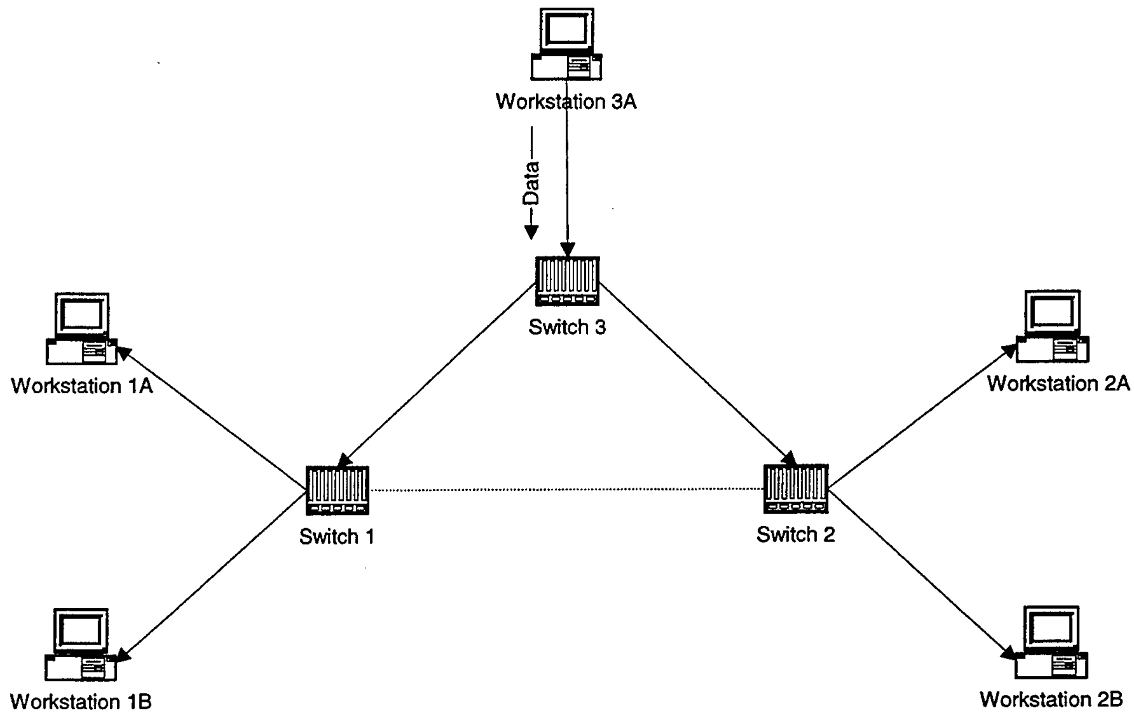

Figure IV-6 Data Transmission from Workstation 3A

The SMART multicast tree will continue to actively reconfigure itself as described above each time that the grant is passed to another workstation. This allows data cells to be transmitted via the most direct route and avoids the unnecessary regeneration and duplication of RM cells and data cells.

B. SMARTER ALGORITHM SUMMARY

The SMARTer algorithm presents a solution to fairness issues that were noted but not addressed by Gauthier et al. [15,16] in their development of the SMART algorithm. This is accomplished through the introduction of a new state variable, the *queue metric*

state variable, into all SMART ports. The *queue metric* state variable is a composite metric based on a SMART workstation's queue size and the oldest cell within its queue. When a grant is passed from one SMART workstation to another, fairness is ensured by having each SMART switch decide where to next send the grant based on relative *queue metric* state variable values.

The SMARTer algorithm introduces an additional variable exclusively to the SMART port of each workstation, the *grant hold time*. The *grant hold time* dictates how long a workstation may hold the grant while requests to transmit data from other SMART workstations are outstanding. The grant hold time is heuristically optimized to minimize total cell loss for the VCC.

The SMARTer algorithm optimizes the ATM multipoint-to-multipoint multicast tree by dynamic altering the multicast tree topology. This is done through the selective deactivation and activation of VC links. This dynamic topology allows the SMART algorithm to ensure that data and RM cells proceed along the most direct route while preventing the unnecessary regeneration and transmission of data and RM cells.

V. RESULTS

The intent of this thesis is to demonstrate that the SMART algorithm is capable of providing fair access to multiple VTC sources on a single VCC while providing reasonable utilization of the available bandwidth. A simulation model that implemented the SMART algorithm was developed using the OPNET simulation tool and is included in Appendix A. A simple ATM multipoint-to-multipoint multicast tree consisting of three SMART ATM switches connected in a delta configuration that served five SMART workstations, as shown in Figure IV-1, was modeled. The metrics of concern were cell loss ratio, utilization, and by extension, fairness.

The SMART algorithm allows for signaling in band (in stream with data cells) and out of band (via separate VCCs). For simplicity of implementation, RM cells used for signaling between SMART ports are assumed to travel via a separate VCC. Additionally, the effects of RM cell delay are assumed to be negligible.

A. SOURCE MODELS

VTC data is comprised primarily of video and audio information. Due to the bandwidth intensive nature of video signals, and to lesser degree audio signals, compression techniques are frequently employed to conserve bandwidth at the expense of signal quality [4]. The SMART algorithm is modeled using both simulated audio and video traffic. The audio source simulates encoded human speech using a "talk-spurt" model, while the video source simulates compressed video using a model that emulates the H.263 low-bit-rate video codec.

1. Audio Model

Human speech is characterized by alternating intervals of activity and silence. This is commonly known as the "talk spurt" model, where the periods of talking average from 0.4 sec to 1.2 sec and the periods of silence average from 0.6 to 1.8 sec. Therefore, human speech is well modeled by a two-state process, as shown in Figure V-1.

Transition rates between states are distributed in Poisson fashion. The transition rate out of the silent state is represented by the parameter λ. The transition rate out of the active state is represented by the parameter α. Therefore, the average talk spurt interval is $1/\alpha$ sec in length, while the average silence interval is $1/\lambda$ in length. The probability that a speaker is active, the speaker activity factor, is defined as $\lambda/(\alpha+\lambda)$. The active and silent period durations are well modeled by exponential distributions with means $1/\alpha$ and $1/\lambda$, respectively. Each source generates a constant stream of cells, *V*, when active. [8]

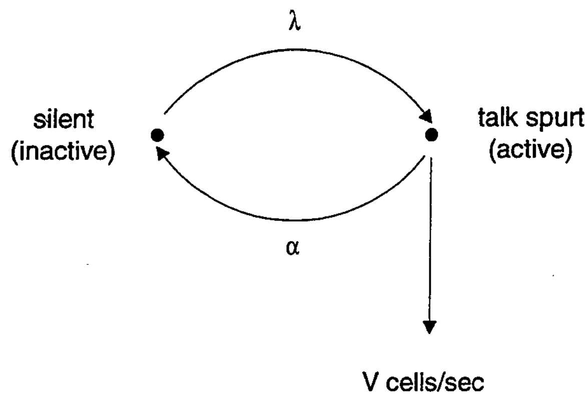

Figure V-1 Talk Spurt Two State Model [8]

Each audio source used in the SMART multicast tree is a talk spurt model. The average silence interval for each source is $1/\lambda = 0.6$ sec in length and the average talk spurt interval is $1/\alpha = 0.4$ sec in length for an activity factor of 0.4. The transmission rate, *V*, for each source is 170 cells per second, which is consistent with ATM transmission rates for encoded telephonic speech [8].

2. Video Model

Although individual units within the IT-21 intranet will contain LANs with backbone capacities of at least 100Mbps and will provide up to 25 Mbps to desktop workstations, bandwidth is limited at the wireless interface connecting individual units. This bandwidth limitation constrains the quality of video available for VTC via that

interface. A reasonable assumption for bandwidth available at the wireless interface is about 1 Mbps [4]. Considering that the wireless interface will support several video sources, and possibly other multimedia sources, simultaneously, a low-bit-rate video source is the most realistic for a tactical VTC application.

Skelly et al. [20], have proposed a model that approximates VBR traffic. The model assumes that a traffic shaping buffer at the video source smoothes cell delivery prior to network entry. Their algorithm models VBR traffic as a deterministic Markov-modulated Rate Process (MMRP). The model uses a discrete-time eight-state Markov chain to capture the histogram of a video sequence. The video sequence is uniformly quantized into eight bins, each corresponding to a different bitrate. Each bitrate, λ_i, is matched to a corresponding state in the Markov chain. When in state i of the Markov chain the model produces one video frame at the respective bitrate λ_i. Any state in the Markov chain can transition to any other state in the Markov chain, as shown in Figure V-2 (several transitions are removed for clarity).

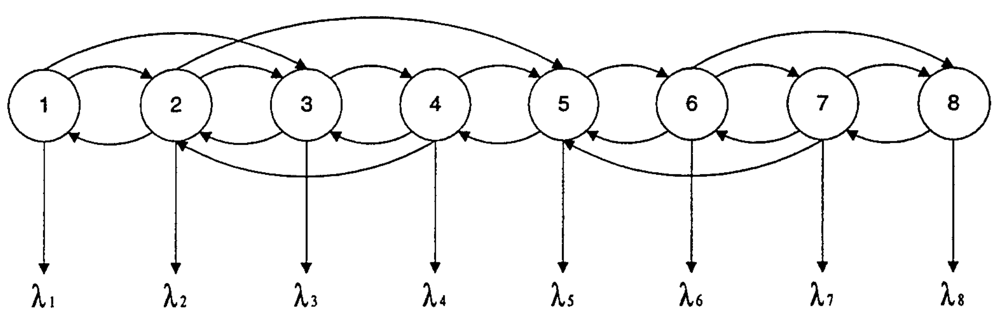

Figure V-2 Markov Chain Model for VBR Video Sources [18]

Parker [18] has shown that Skelly's MMRP model can reasonably model an ITU standard H.263 low-bit-rate video stream using only six states. Parker's model is used for each video source in the SMART multicast tree. The average transmission rate, λ_{avg}, for each source is 212.21 cells per second and the peak transmission rate, λ_{max}, for each source is 632.95 cells per second.

B. SIMULATION SCENARIOS

Simulation scenarios consisted of two major groups with five subgroups each. The type of source used at each workstation, audio or video determined the major groups. The number of sources transmitting, one through five, determined the five subgroups. Additionally, various permutations within a subgroup were examined. For example, for both the audio and video source cases, the three-source subgroup was further divided with respect to what specific workstations were transmitting. The case where workstations 1A, 2A, and 3A were all transmitting was contrasted with the case where workstations 1A, 1B, and 2A were all transmitting. This was done to determine whether the results would vary based on the different configurations of the SMART switches (SMART switch 3 serves only one workstation while the other SMART switches each serve two workstations).

Audio and video simulations were run for simulation times of 2000 and 4000 seconds, respectively. Further, each simulation was run a minimum of five times each using different simulation kernel seeds in order to confirm consistency of results. Each subgroup was examined to heuristically determine the point at which the total cell loss for all workstation queues dropped to a minimum value. This was accomplished by varying the available bandwidth on the VCC for each simulation run until the desired datum point was reached.

Most video applications experience an unacceptable degradation in quality at error rates of greater than 10^{-6} [21]. Therefore, a maximum acceptable cell loss rate (CLR) of 10^{-6} was originally chosen for all simulation scenarios. However in all scenarios, given the constant simulation run times stated above, CLR promptly drops to zero as the limit of 10^{-6} is approached from a CLR of approximately 3×10^{-6}. As a result, longer simulation run times would be required to determine the point at which CLR actually drops below 10^{-6}. This would require a prohibitive amount of both real time and computer memory to accomplish. Consequently VCC bandwidth was varied in each case to achieve a total cell loss of zero for each simulation run.

Additional simulations were run in which available VCC bandwidth was reduced to the point where each transmitting workstation queue was forced to experience significant cell loss. This was done in order to compare the fairness provided by the SMART multicast tree to each transmitting workstation, as it would be expected that if the algorithm were truly fair, all queues for workstations that were transmitting would have roughly the same cell loss rate. These simulations were run on a sufficient number of workstation subgroup permutations within each group (video and audio) so that the fairness of the SMART algorithm could be effectively evaluated.

C. SIMULATION RESULTS

1. VCC Utilization Using Video Sources

The results for VCC utilization using only video sources are shown in Figure V-3. VCC utilization was calculated by multiplying the average cell rate per source by the number of sources and then dividing by the capacity of the VCC. VCC utilization for one video source was determined in order to establish a baseline for comparison with multiple video source cases. The single video source VCC utilization of 0.4129 is fairly reasonable considering that the goal was to drive cell loss to zero and that the peak cell rate for the video source is roughly three times the average cell rate.

As the number of transmitting video sources increases, VCC utilization increases. This is due to the deterministic smoothing at each video source and the selective manner in which the grant is passed. The deterministic smoothing reduces the bursty nature of the video source. The grant is preferentially passed to the queue that has the largest occupancy with the oldest data cells. This allows the source that is currently transmitting at the highest cell rate to receive preferential treatment over the other sources. Although a source with a temporarily higher rate does not receive preferential treatment to the point that the other sources are "locked out" of the VCC, it is allocated the majority of the bandwidth available for that period of time. If the sources are all bursty in nature and their periods of burstiness are relatively short and independent of one another, then it is unlikely that the majority of the sources will each transmit at their respective peak cell

rates (PCRs) simultaneously. This allows the available VCC bandwidth to be more efficiently matched to the bandwidth demands of the sources, as the majority of the VCC bandwidth is selectively allocated to the source with the greatest bandwidth need. This effectively provides a time division multiplexing gain to the multicast tree.

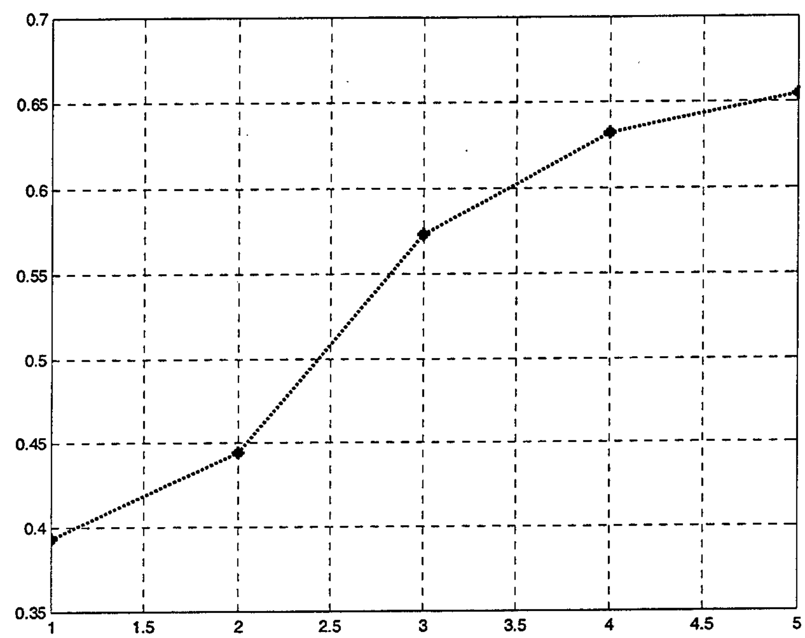

Figure V-3 Utilization for Video Sources

A comparison was made between the case when sources 1A, 2A, and 3A were transmitting and when sources 1A, 1B, and 2A were transmitting. This was done to determine whether there was any difference in VCC utilization when each workstation has a dedicated SMART switch available to it and when two workstations must share one SMART switch. There was no difference in utilization between the two cases.

2. VCC Utilization Using Audio Sources

The results for VCC utilization using only audio sources are shown in Figure V-4. VCC utilization was calculated by multiplying the active cell rate per source by the

number of sources and the speaker activity factor and then dividing by the capacity of the VCC. VCC utilization for one audio source was determined in order to establish a baseline for comparison with multiple audio source cases. The single audio source VCC utilization of 0.4121 is fairly reasonable considering that the goal was to drive cell loss to zero and that the peak cell rate for the audio source is 2.5 times the average cell rate.

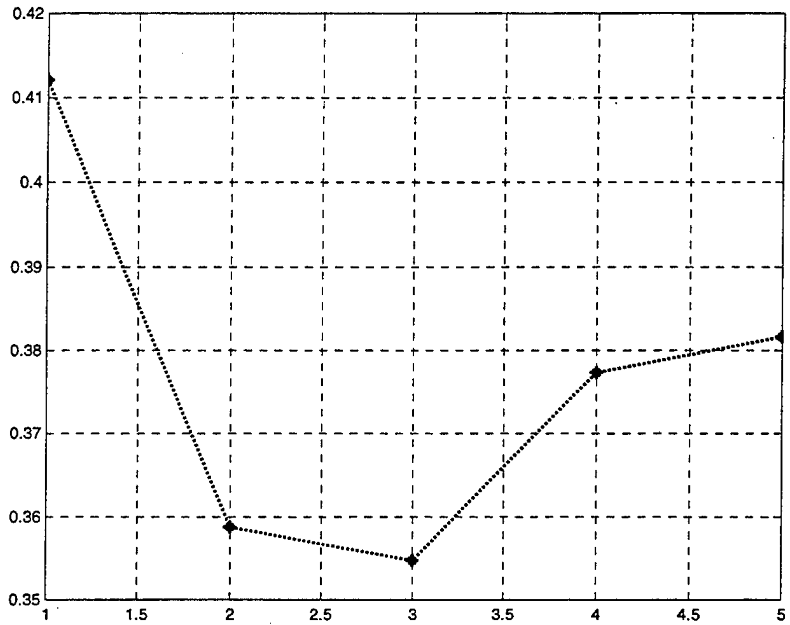

Figure V-4 Utilization for Audio Sources

As can be seen from Figure V-4, VCC utilization drops significantly when more than one source is active on the VCC. Although some evidence of time division multiplexing gain can be seen as the number of sources increases from two to five, the increase in utilization is relatively small, especially when compared to the utilization of the single source case. The reason between the disparity in utilization between the audio source case and the video source case has to do with the burstiness of the respective sources. Although a video source is typically more bursty than an audio source, the video source used here has been deterministically smoothed. Therefore, the audio source is

actually more bursty than the video source, as can be seen from Figure V-5 and Figure V-6.

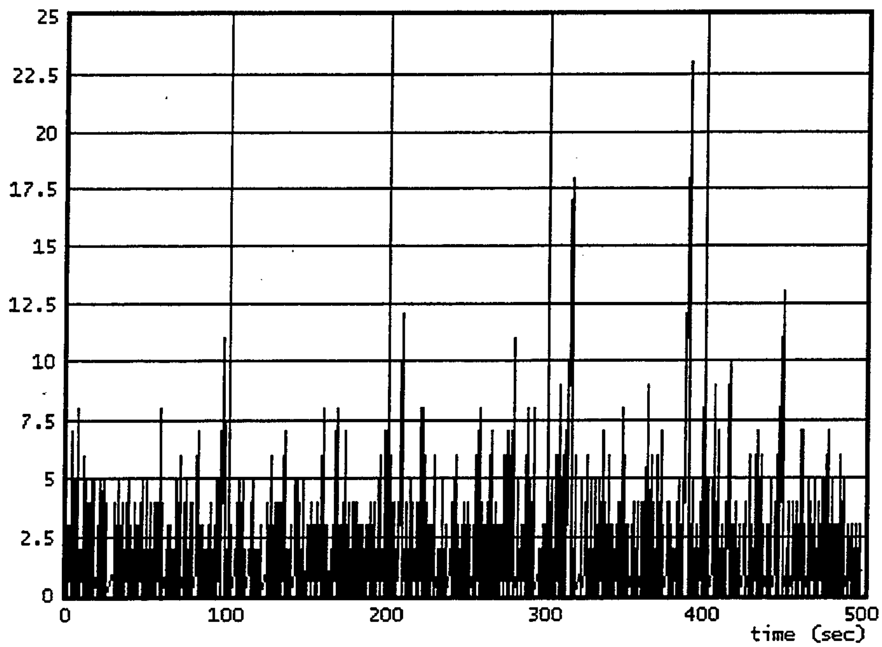

Figure V-5 Queue Activity for Single Audio Source (Utilization = 0.4121)

Since the audio sources are more bursty, less gain is derived from the effects of time division multiplexing. This is due to the greater likelihood that more than one source will experience a burst period at the same time, thus requiring greater bandwidth to avoid excessive cell loss. Additionally, the audio source state is binary in that it is either transmitting at a given cell rate or not transmitting at all. The video source transmits at various cell rates, the least of which is something greater than zero. As a result, the VCC will always have a cell flow, however small, in the video case, while the VCC will have periods of no cell activity for the audio case. Therefore the level of VCC utilization during periods of minimal VCC utilization is higher for the video case than the audio case and as such contributes to overall VCC utilization being greater in the video case than the audio case.

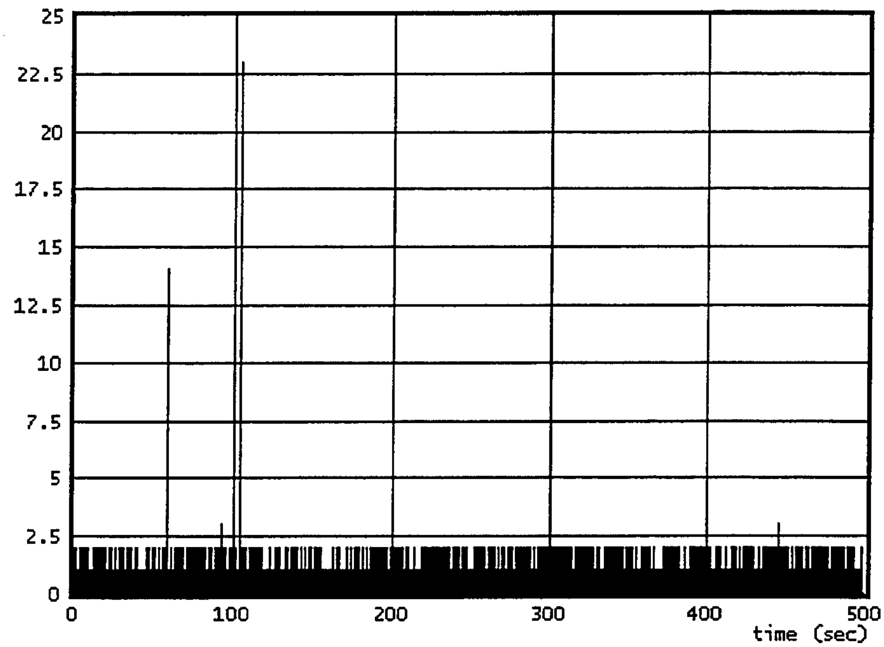

Figure V-6 Queue Activity for Single Video Source (Utilization = 0.3930)

3. Fairness

Fairness in passing the grant between workstations was evaluated based on comparative cell loss ratios when the workstation queues were forced to experience excessive cell loss. Fairness was evaluated in the audio example for three cases: workstations 1A, 1B, 2A, and 2B active; workstations 1B, 2A, 2B, and 3A active; and all workstations active. In each case the difference in cell loss between the workstations served by SMART switches 1 and 2 was negligible. Specifically, for the first case the average cell losses were 0.094, 0.100, 0.093, and 0.100, respectively, after 8 simulation runs using different kernel seeds for each run. It appears that there may be some slight favoritism toward the "A" workstations, as their cell loss is slightly lower than that for the "B" workstations. This is most likely due to the fact that when a choice between workstations is to be made when passing the grant, if the *queue metrics* are equal, the grant will be passed to the "A" workstation. This is easily solved by either randomly choosing a workstation or by alternating between each workstation when the *queue*

metrics are equal. The former choice would be preferable, as the latter requires additional memory, the size of which would grow with the number of workstations served.

The average cell losses for each workstation in the second case of forced cell loss were 0.075, 0.078, 0.079, 0.076, and 0.104, respectively. The cell losses for workstations 1A, 1B, 2A, and 2B were roughly comparable, while the cell loss for workstation 3A was significantly (35 percent) higher than the average cell loss of the other workstations. The third case of forced cell loss produced similar results. The reason for this disparity is probably related to the fact that SMART switch 3 serves only one workstation, while the other SMART switches serve two workstations. The SMART switches serving two workstations receive approximately twice as many requests to transmit data from the workstations they serve, and this may bias the passing of the grant.

The disparity in cell loss noted above was solved be modifying the *queue metric* state variable for workstation 3A through the introduction of a *boost* variable. The *boost* variable was used to increase the magnitude of the *queue metric* state variable for workstation 3A relative to the other workstations. Boosting the *queue metric* state variable by a factor of approximately 1.2 and 2.0 for video and audio sources, respectively, reduced the disparity in forced cell loss to negligible levels. This may indicate the need to dynamically modify the *queue metrics* for each workstation based on the number of active workstations a SMART switch serves.

4. Overhead

RM cell activity was monitored for both the video and audio examples. In each case all five sources were actively transmitting, VCC bandwidth was set for maximum utilization, and the *boost* variable was set to 1.0 (i.e., the *queue metric* state variable for workstation 3A was not preferentially biased). The RM cell activity is defined as the ratio of the total number of RM cells received to the total number of data cells received by a SMART switch or SMART workstation. RM cell activity ratios are shown in Table V-1.

SMART Switch/Workstation	Cell Activity Ratio	
	Video	Audio
1A	0.29	0.81
1B	0.29	0.80
2A	0.30	0.80
2B	0.30	0.78
3A	0.22	0.74
1	1.37	3.44
2	1.39	3.44
3	0.76	2.32

Table V-1 RM Cell Activity Ratios

As shown in Table V-1, the RM cell activity ratios for the audio case are about two to three times larger than for the video case. This can be attributed, at least in part, to the fact that the audio sources have silent periods. If a source holding the grant goes silent, the grant will be passed to another source requesting to send data, regardless of whether the *grant hold time* has been exceeded. Since this will cause the grant to be passed among the SMART workstations with a greater frequency, it results in a greater number of RM cells being transmitted.

RM cell activity for the SMART switches is about three to four times that of the SMART workstations. The SMART switches in this example receive RM cells from two SMART switches and either one or two SMART workstations, whereas the SMART workstations receive RM cells only from their respective SMART switch. Therefore, it is expected that the SMART switches have higher cell activity ratios than the SMART workstations.

The SMART algorithm was modeled using the delta configuration shown in Figure IV-1. Simulations were conducted using audio source models and video source models separately. In each case it was demonstrated that access to the VCC could be

fairly granted to each SMART workstation. However, in order to ensure fairness it was necessary to introduce a *boost* variable to bias the *queue metric* state variable for workstation 3A. This was done because the SMART algorithm appears biased towards SMART switches that serve a larger number of active sources.

VCC utilization for a single active source was 0.4121 and 0.3930 for audio and video sources, respectively. VCC utilization increased to 0.6542 as the number of video sources was raised to five, while VCC utilization dropped to 0.3588 and raised to 0.3816 as the number of audio sources was raised to five. The video case benefited from the effects of time division multiplexing gain, which was possible largely due to the traffic smoothing at the video source. The audio case saw no such gain, primarily because the audio sources were much burstier in nature than the video sources.

The overhead due to RM cell activity is significant, as can be seen in Table V-1. SMART switches receive considerably more RM cells than SMART workstations, which is not surprising considering that each SMART switch interacts with considerably more SMART ports than each SMART workstation. The audio case produces relatively more RM cells than the video case. This is likely due to the grant being passed among SMART workstations with a greater frequency than in the video case.

VI. CONCLUSIONS

A. SUMMARY OF WORK

VTC is a powerful subset of the multimedia capabilities available to the US Navy as it develops the IT-21 Intranet to implement its warfare paradigm shift from platform-centric to network-centric. A primary component of the IT-21 Intranet will be ATM LANs that will eventually provide ATM services to the desktop. VTC applications require that the underlying application and network layers provide multipoint-to-multipoint multicast capability. A specification for ATM multipoint-to-multipoint multicast does not currently exist, and, as such, ATM does not support multipoint-to-multipoint VTC applications without an intermediate protocol layer.

Gauthier et al. [15,16], have proposed an algorithm, the SMART protocol, which lies entirely within the ATM layer and provides multipoint-to-multipoint multicast services over one or more VCCs. However, the SMART algorithm does not specify how fairness of access is ensured for multiple sources operating simultaneously on a multipoint-to-multipoint multicast tree.

This work produced a modified, or "SMARTer," SMART algorithm to provide fair access to multiple sources over multipoint-to-multipoint tree using a single VCC. The SMARTer algorithm employs three additional variables to ensure fairness of access: the *queue metric* state variable, the *grant hold time* variable, and the *boost* variable. The *queue metric* is a composite state variable that combines a workstation's queue size and the cell age of the oldest data cell in that queue to determine to which source to next pass the grant. The *grant hold time* specifies how long a source can hold the grant. The *boost* variable biases a workstation's *queue metric* as necessary to ensure fairness. The ability of the SMARTer algorithm to provide fair access for scenarios with either audio sources or video sources was demonstrated using the OPNET simulation tool. The multipoint-to-multipoint multicast tree configuration used for the simulation was that shown in Figure IV-1. Simulations were run for both the audio and video cases. In each case the number

of active (workstation) sources transmitting to the other workstations on the multipoint-to-multipoint multicast tree was incrementally varied from one to five.

VCC utilization was also examined for scenarios using the audio and video source configurations described above. VCC utilization improved for the video case as the number of sources was increased. This was due to the time division multiplexing gain experienced by the traffic smoothed video sources. VCC utilization for the audio case dropped significantly as a second source was added and then slowly increased as the number of sources was raised to five. The relatively high bursty nature of the audio sources prevented gains comparable to those of the video case.

Finally, RM cell activity was examined for the above configurations. The amount of RM cell activity relative to data cell activity was significant. Greater relative RM cell activity was seen for the audio case than the video case and is likely due to the greater frequency with which the grant is passed between workstations in the audio case. SMART switches received a greater relative amount of RM cells than SMART workstations, which is to be expected considering that each SMART switch is linked to more potential sources of RM cells than is each SMART workstation.

B. SUGGESTIONS FOR FUTURE RESEARCH

The intent of this work was to demonstrate that the SMART algorithm could be implemented to fairly provide access to multiple users on a multipoint-to-multipoint multicast tree. The RM cell plays a fundamental role in the operation of the SMART algorithm and is instrumental in ensuring fairness and maximizing VCC utilization. For example, in order for the grant to be passed as soon as possible and to ensure that the grant is passed to the correct workstation, state variable data must be updated frequently, which is done through RM cells. However, the number of RM cells generated directly impacts the overhead required to operate the SMART algorithm. Although this work has demonstrated that the overhead due to RM cells can be significant, little effort was made to minimize the production of RM cells. An enlightening endeavor would be to attempt

to minimize the number of RM cells produced by the SMART algorithm while both ensuring that fairness is maintained and maximizing VCC utilization.

The audio source used for this work, the "talk-spurt" model, does not take into account the impact of additional sources on the original source. For example, if one person is speaking and is interrupted by another, it is rare that both continue to talk. Usually one person will yield until the other finishes speaking. By way of extension, it is extremely unlikely that several individuals will talk simultaneously during a VTC. The "talk-spurt" model does not take this into account and as such is a worst-case model. A model that more closely represents each source's activity in a scenario where there are multiple sources would be appropriate for the audio scenarios examined here. Such a model would very likely reveal improvements in VCC utilization over the results presented in this work for the audio scenarios.

APPENDIX A. OPNET MODEL CODE

This section contains the OPNET node and process models used to generate the simulation results shown in Chapter V. The node model contains several process models, each of which consists of finite state machines and associated code segments.

A. SMART ATM DELTA CONFIGURATION NODE MODEL

The node model used to implement the SMART ATM multipoint-to-multipoint multicast delta configuration as originally discussed in Chapter IV and shown in Figure IV-1 is illustrated in Figure A-1.

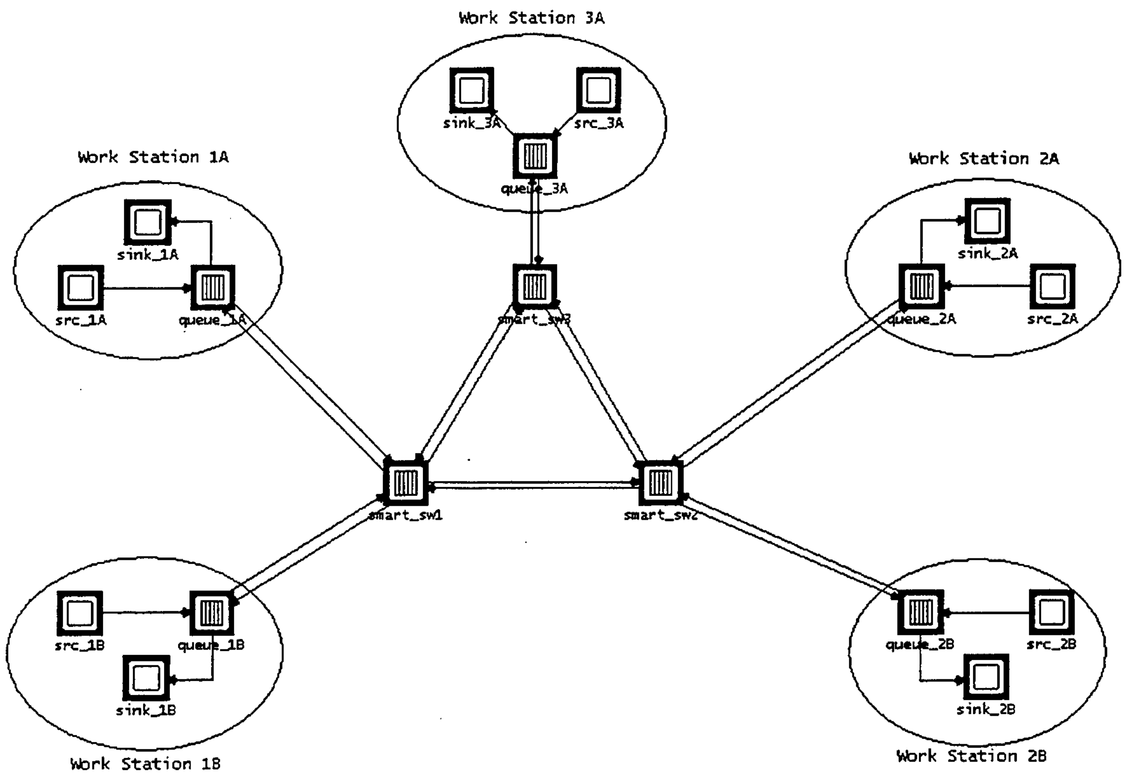

Figure A-1 OPNET Implementation of Delta Configuration SMART Multicast Tree

B. AUDIO SOURCE MODEL

The OPNET model for the "talk-spurt" two state audio source discussed in chapter V is shown in Figure A-2. All code shown for the "init," "silence," and "active" states is located in the "Enter Execs" block of each state.

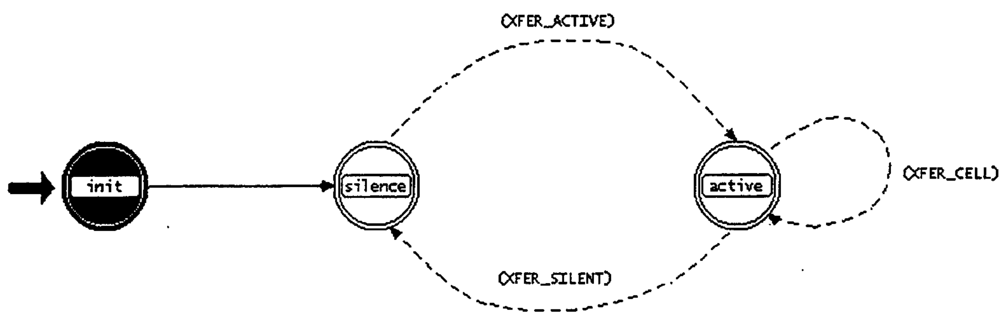

Figure A-2 Finite State Machine for Two-State Audio Model

1. Header Block

```
OPC_COMPILE_CPP

#include "iostream.h"
#include "ams_pk_support.h"

/* Define op_codes for self_intrpt's */
#define           VOICE_ACTIVE 1
#define           VOICE_SILENT 2
#define           XMIT_CELL 3

/* Define macro's for transition states */
#define           XFER_ACTIVE op_intrpt_code()==VOICE_ACTIVE
#define           XFER_SILENT op_intrpt_code()==VOICE_SILENT
#define           XFER_CELL op_intrpt_code()==XMIT_CELL

AtmT_Cell_Header_Fields*      set_header(int);
```

2. State Variables Block

```
int               \cell_size;
int               \flag;
int               \pk_count;
int               \source_id;

double            \inv_lambda;
```

```
double              \inv_alpha;
double              \cell_gen;
double              \V;

Stathandle          \pk_cnt_stathandle;

Distribution *      \active_dist;
Distribution *      \silent_dist;
```

3. Temporary Variables Block

```
double                        silent_rv;
double                        active_rv;

Packet*                       new_packet;

AtmT_Cell_Header_Fields*      atm_hdr_ptr;
```

4. Init State

```
/* Initialize Variables */
inv_lambda = 0.6;/*0.6*/
inv_alpha = 0.4;/*0.4*/
V = 170.0;
cell_size = 53*8;
flag = 0;

/* Load Distributions */
active_dist = op_dist_load("exponential",inv_lambda,0.0);
silent_dist = op_dist_load("exponential",inv_alpha,0.0);

/* Set cell generation interval */
cell_gen = 1.0/V;
pk_count = 0;
pk_cnt_stathandle = op_stat_reg("packet
count",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
```

5. Silence State

```
/* Clear any ongoing scheduled self interupts (i.e. cell gen) */
op_intrpt_clear_self();
/*pk_count = 0;*/

/* Initiate random variable to xfer to active state */
active_rv = op_dist_outcome(active_dist);

/* Schedule self interupt */

op_intrpt_schedule_self(op_sim_time() + active_rv, VOICE_ACTIVE);

/* Reset flag to zero */
```

```
flag = 0;
```

6. Active State

```
/* Generate Cell */
new_packet = op_pk_create_fmt("rrw_atm_cell");
/* Allocate memory for the header and assign PT value for data cell .
*/
atm_hdr_ptr = set_header(0);

/* Load the ATM header. */
op_pk_nfd_set(new_packet,"header
fields",atm_hdr_ptr,op_prg_mem_copy_create,\
               op_prg_mem_free,sizeof(AtmT_Cell_Header_Fields));

/* Load the time value into the data field so that overall delay may be
computed. */
op_pk_nfd_set(new_packet,"Time_Start",op_sim_time ());

/* Send Cell */
op_pk_send_forced(new_packet,0);
pk_count = pk_count + 1;
op_stat_write (pk_cnt_stathandle, pk_count);

/* Scheduled interupts for each cell that is created */
op_intrpt_schedule_self(op_sim_time() + cell_gen,XMIT_CELL);

/* Schedule only one exponential self interupt per active state */
if(!flag)

      {
      /* Initiate rand var for use in sched intrpt for xfer to silent
state */
      silent_rv = op_dist_outcome(silent_dist);

      /* Schedule self intrpt to xfer to silent state */
      op_intrpt_schedule_self(op_sim_time() + silent_rv, VOICE_SILENT);

      /* Set flag to one to indicate silent_rv self intrpt has been
initiated */
      flag = 1;
      }
```

C. VIDEO SOURCE MODEL

The OPNET model for the deterministic MMRP video (VBR) source discussed in chapter V is shown in Figure A-2. This OPNET model was developed by Parker [18], with minor modifications by the author for implementation in this work. All code shown

for the “init,” “transition,” and “send_cell” states is located in the “Enter Execs” block of each state.

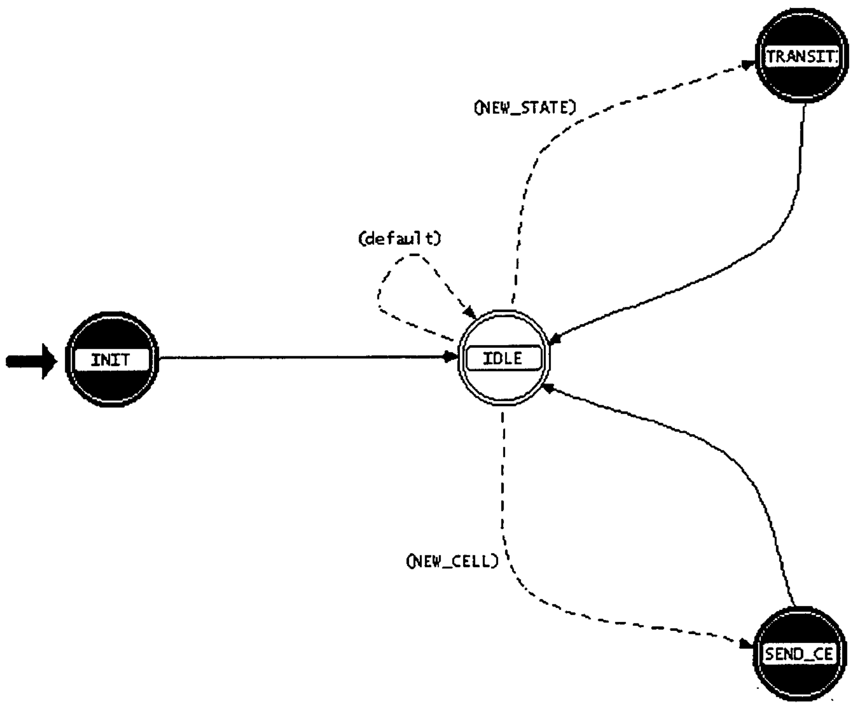

Figure A-3 Parker’s Finite State Machine for a Video Traffic Model [18]

1. Header Block

```
#include  "ams_pk_support.h"

#define SEND_CELL    0
#define CHANGE_STATE 10
#define MAX_SOURCE 7

#define NEW_STATE   ((op_intrpt_type() == OPC_INTRPT_SELF) &&\
                     (op_intrpt_code() >= CHANGE_STATE))

#define NEW_CELL    ((op_intrpt_type() == OPC_INTRPT_SELF) &&\
                     ((op_intrpt_code() >= SEND_CELL) &&\
                      (op_intrpt_code() <= (SEND_CELL + MAX_SOURCE))))

#define INF         9999999999
#define VCI_BASE    100
```

2. State Variables Block

```
Objid \self_id;

int    \curr_state[MAX_SOURCE];
int    \pk_count;
int    \next_state[MAX_SOURCE];
int    \sources;

double       \transit_time;
double       \interval;

Distribution**    \state_dist;

Stathandle  \state0_shandle;
Stathandle  \state1_shandle;
Stathandle  \rate_shandle;

Evhandle     \cell_intrpt[MAX_SOURCE];

Stathandle  \pk_cnt_stathandle;
```

3. Temporary Variables Block

```
double M[6][6] = {{0.000,1.807,0.636,0.153,0.025,0.000},\
                  {1.240,0.000,0.288,0.399,0.044,0.022},\
                  {5.667,0.833,0.000,0.167,0.000,0.000},\
                  {2.800,3.920,0.280,0.000,0.000,0.000},\
                  {7.000,0.000,0.000,0.000,0.000,0.000},\
                  {0.000,7.000,0.000,0.000,0.000,0.000}};
double lambda[6] = {132.82,232.85,332.87,432.90,532.92,632.95};

Packet* cell_ptr;

int ix;
int jx;
int source_id;

AtmT_Cell_Header_Fields* atm_hdr_ptr;
```

4. Init State

```
/* get source module's own object id */
self_id = op_id_self();

/* get the requested number of multiplexed video sources */
op_ima_obj_attr_get (self_id, "Number_of_Sources", &sources);

/* allocate space and load distributions */
state_dist =
(Distribution**)(op_prg_mem_alloc(sizeof(Distribution*)*36));
```

```
for (ix=0;ix<6;ix++){
   for (jx=0;jx<6;jx++){
      if (M[ix][jx]>0.0){
         state_dist[ix*6+jx] =
op_dist_load("exponential",1.0/M[ix][jx],0);
      }
      else{
         state_dist[ix*6+jx] = op_dist_load("exponential",INF,0);
      }
   }
}

/* generate an initial interupt for each source, arbitrarily */
/* choosing the 0th state.                                    */
for (ix = 0;ix < sources;ix++){
   next_state[ix] = 0;
   op_intrpt_schedule_self(op_sim_time() + .000001,CHANGE_STATE + ix);
}

// Troubleshooting statistics
state0_shandle =
op_stat_reg("State0",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
state1_shandle =
op_stat_reg("State1",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);

pk_count = 0;
pk_cnt_stathandle = op_stat_reg("packet
count",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
```

5. Transition State

```
/* One of the sources is changing state; get the source's id.  */
source_id = op_intrpt_code() - CHANGE_STATE;
/*printf("Source ID: %d",source_id);*/

/* Cancel the pending cell transmission self-interupt for this source.
*/
if (op_ev_valid(cell_intrpt[source_id])){
   op_ev_cancel(cell_intrpt[source_id]);
}


/* Assign the new current state.  */
curr_state[source_id] = next_state[source_id];

/* Find next state and transition time */
next_state[source_id] = 0;
transit_time = op_dist_outcome(state_dist[curr_state[source_id]*6]);

/* Search for the shortest time, this is the next state. */
for (ix = 1;ix < 6;ix++){
```

```
    interval = op_dist_outcome(state_dist[curr_state[source_id]*6 +
ix]);

    if (interval < transit_time){
        transit_time = interval;
        next_state[source_id] = ix;
    }
}

/* Send a cell now and schedule next departure */
cell_ptr = op_pk_create_fmt("rrw_atm_cell");

/* Load the time value into the data field so that overall delay may be
computed. */
op_pk_nfd_set(cell_ptr,"Time_Start",op_sim_time ());

/* Allocate memory for the header and assign fields.  */
atm_hdr_ptr = (AtmT_Cell_Header_Fields*)
op_prg_mem_alloc(sizeof(AtmT_Cell_Header_Fields));
atm_hdr_ptr->VCI = VCI_BASE + source_id;
/*printf("\nSource VCI: %d",atm_hdr_ptr->VCI);*/

/* Load the ATM header and transmit the cell. */
op_pk_nfd_set(cell_ptr,"header
fields",atm_hdr_ptr,op_prg_mem_copy_create,\
              op_prg_mem_free,sizeof(AtmT_Cell_Header_Fields));
op_pk_send(cell_ptr,0);

pk_count = pk_count + 1;
op_stat_write (pk_cnt_stathandle, pk_count);

cell_intrpt[source_id] = op_intrpt_schedule_self(op_sim_time() +
1.0/lambda[curr_state[source_id]],\
                         SEND_CELL + source_id);

/* Schedule state transition */
op_intrpt_schedule_self(op_sim_time() + transit_time,CHANGE_STATE +
source_id);

// Troubleshooting section
/*if (source_id == 0){
    op_stat_write(state0_shandle,curr_state[source_id]);
}
else{
    op_stat_write(state1_shandle,curr_state[source_id]);
}*/
```

6. Send_Cell State

```
/* One of the sources is changing state; get the source's id.  */
source_id = op_intrpt_code() - SEND_CELL;
printf("Source ID (Cell): %d",source_id);
```

```
/* Create and send an unformatted cell.  */
cell_ptr = op_pk_create_fmt("rrw_atm_cell");

/* Load the time value into the data field so that overall delay may be
computed. */
op_pk_nfd_set(cell_ptr,"Time_Start",op_sim_time ());

/* Allocate memory for the header ans assign fields.  */
atm_hdr_ptr = (AtmT_Cell_Header_Fields*)
op_prg_mem_alloc(sizeof(AtmT_Cell_Header_Fields));
atm_hdr_ptr->VCI = VCI_BASE + source_id;
printf("\nSource VCI (Cell): %d",atm_hdr_ptr->VCI);

/* Load the ATM header and transmit the cell. */
op_pk_nfd_set(cell_ptr,"header
fields",atm_hdr_ptr,op_prg_mem_copy_create,\
              op_prg_mem_free,sizeof(AtmT_Cell_Header_Fields));
op_pk_send(cell_ptr,0);

pk_count = pk_count + 1;
op_stat_write (pk_cnt_stathandle, pk_count);

/* Schedule next cell departure. */
cell_intrpt[source_id] = op_intrpt_schedule_self(op_sim_time() +
1.0/lambda[curr_state[source_id]],\
                        SEND_CELL + source_id);

// Troubleshooting section
/*
if (source_id == 0){
   op_stat_write(state0_shandle,curr_state[source_id]);
}
else{
   op_stat_write(state1_shandle,curr_state[source_id]);
}
*/
```

D. OPNET SINK MODEL

The OPNET sink model provided by MIL3, as modified by the author, is shown in Figure A-4. All code shown for the "init" and "discard" states is located in the "Enter Execs" block of each state.

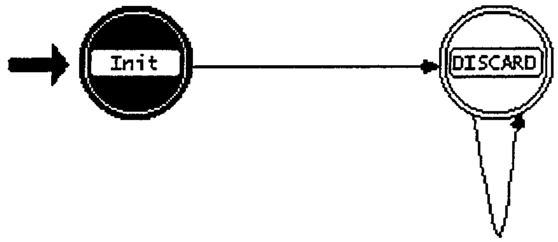

Figure A-4 OPNET Finite State Machine for a Sink

1. Header Block

```
OPC_COMPILE_CPP

#include "iostream.h"
```

2. State Variables Block

```
Packet *                            \cell_ptr;

double                             \Time_Start;
double                               \delay;

Stathandle                         \delay_stat;
```

3. Init State

```
delay_stat = op_stat_reg("End-to-End Delay",OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
```

4. Discard State

```
cell_ptr = op_pk_get (op_intrpt_strm ());

/* Get time of origin and compute delay. */
op_pk_nfd_get(cell_ptr,"Time_Start",&Time_Start);
delay = op_sim_time()- Time_Start;

op_stat_write(delay_stat,delay);
op_pk_destroy (cell_ptr);
```

E. SMART WORKSTATION MODEL

The OPNET model for the SMART workstation discussed in chapters IV and V is shown in Figure A-5. All code shown for all states except for the idle state is located in

the “Enter Execs” block of each state. Code for the idle state is located in the “Exit Execs” block for that state.

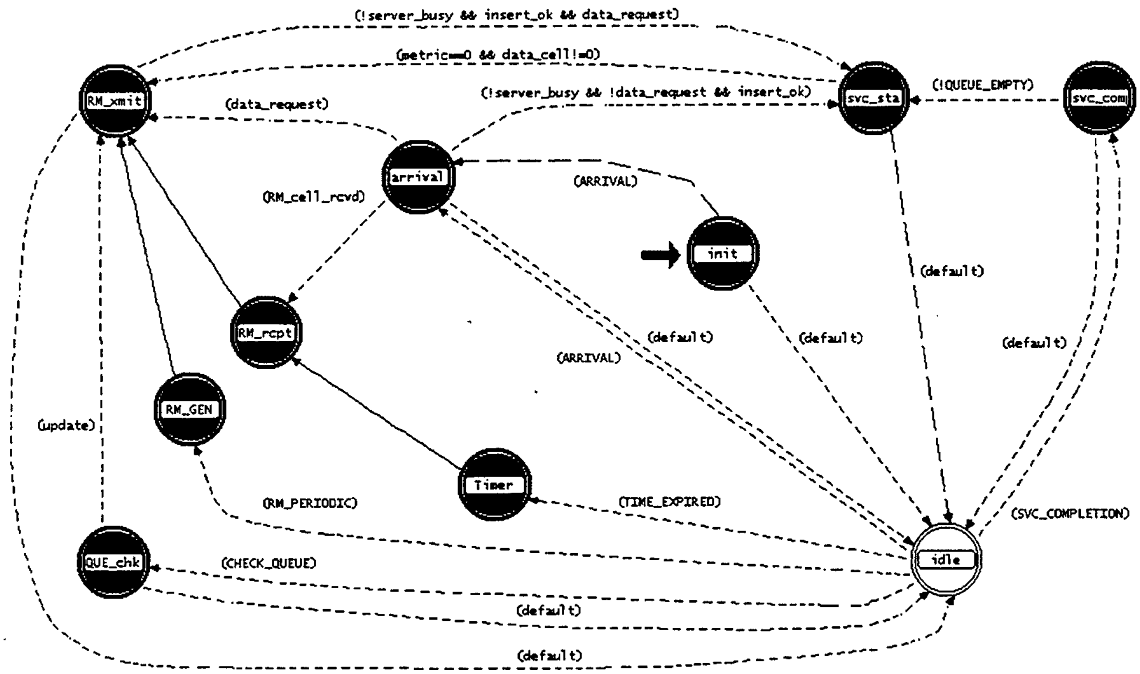

Figure A-5 Finite State Machine for a SMART Workstation

1. Header Block

```
OPC_COMPILE_CPP

#include "iostream.h"
#include "ams_pk_support.h"

/* Define constants. */
#define GRANT_time                .010
#define MAX_DELAY                .1475
#define RM_mult                    13
#define RM_flag                     6
#define QUEUE_SIZE                100
#define ACTIVE                      1
#define INACTIVE                    0
#define CANCEL                      1
#define SOURCE_time             1/170

/* Set 'port' array size. */
/* Determined by number of input ports. */
#define Port_Sz                     1
```

```
/* Define op codes for self interrupts. */
#define CELL_flag_xmit              0
#define RM_flag_recd                1
#define GRANT_flag                  2
#define RM_flag_self                3
#define Queue_Check                 4

/* Define macros for transition states. */
#define QUEUE_EMPTY      (op_q_empty ())

#define ARRIVAL          op_intrpt_type () == OPC_INTRPT_STRM
#define SVC_COMPLETION   (op_intrpt_code () == CELL_flag_xmit) &&\
                         (op_intrpt_type () == OPC_INTRPT_SELF)
#define RM_PERIODIC(op_intrpt_code () == RM_flag_self) &&\
                         (op_intrpt_type () == OPC_INTRPT_SELF)
#define TIME_EXPIRED(op_intrpt_code () == GRANT_flag) &&\
                         (op_intrpt_type () == OPC_INTRPT_SELF)
#define CHECK_QUEUE(op_intrpt_code () == Queue_Check) &&\
                         (op_intrpt_type () == OPC_INTRPT_SELF)


/* Define struct function. */
AtmT_Cell_Header_Fields* set_header(int);
```

2. State Variables Block

```
int          \server_busy;
int          \cell_total;
int          \cell_destroy;
int          \cell_queue;
int          \RM_cell_rcvd;
int          \rm_rcvd_cnt;
int          \pk_count;
int          \source_flag;
int          \canx_flag;
int          \fn_objid;
int          \stream_id;
int          \num_fixed_nodes;
int          \SMART_neighbor;
int          \SMART_value;
int          \flag_RM_immediate;
int          \insert_ok;
int          \accept_grant[Port_Sz];
int          \accept_request[Port_Sz];
int          \sent_grant[Port_Sz];
int          \sent_request[Port_Sz];
int          \sent_seq_num[Port_Sz];
int          \recvd_grant[Port_Sz];
int          \recvd_request[Port_Sz];
int          \recvd_seq_num[Port_Sz];
int          \bias[Port_Sz];
int          \status[Port_Sz];
```

```
int          \timer_expired;
int          \source_id;
int          \temp;
int          \link;
int          \data_cell;
int          \data_request;
int          \grant_hold;
int          \metric;
int          \update;
int          \delay_metric;

int *        \status_ptr;

double       \rm_cell_gen_time;
double       \rm_cell_sched_time;
double       \service_rate;
double       \cell_loss;
double       \pk_svc_time;
double       \queue_timer;

Packet *     \pkptr_1;

Objid        \own_id;

Stathandle   \cell_loss_prob;
Stathandle   \cell_queue_stat;
Stathandle   \rcvd_rm_cell_act;
Stathandle   \pk_count_stat;
Stathandle   \grant_hold_stat;

Evhandle     \evh_1;
Evhandle     \evh_2;
```

3. Temporary Variables Block

```
int                             i;
int                             payload_type;
int                             stream_id;
int                             accept_flag;
int                             rm_flag;
int                             send_flag;
int                             ssn_flag;

double                          pk_len;
double                          Time_Start;
double                          delay;

AtmT_Cell_Header_Fields*        atm_hdr_ptr;

Packet *                        cell_ptr;
Packet *                        pkptr_2;
Packet *                        pkptr_3;
```

4. Init State

```
/* initially the server is idle */
server_busy = 0;
rm_rcvd_cnt = 0;
pk_count =    0;

/* Get queue module's own object id. */
own_id = op_id_self ();

/* Get assigned value of server processing rate. */
op_ima_obj_attr_get (own_id, "service_rate", &service_rate);

/* Get source ID. */
op_ima_obj_attr_get (own_id, "source_id", &temp);

/* set dummy SMART value */
op_ima_obj_attr_get (own_id, "SMART_value", &SMART_value);

pk_svc_time = 1.0/service_rate;
rm_cell_gen_time = RM_mult*pk_svc_time;

/* Initialize state variables */
metric          = 0;
data_cell       = 0;
cell_total      = 0;
cell_destroy    = 0;
cell_queue      = 0;
cell_loss       = 0;
canx_flag       = 0;
grant_hold      = 0;
cell_loss_prob  = op_stat_reg("Cell Loss Probability",\
            OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
cell_queue_stat = op_stat_reg("Queue Size Stat",OPC_STAT_INDEX_NONE,\
            OPC_STAT_LOCAL);
grant_hold_stat = op_stat_reg("Grant Hold",OPC_STAT_INDEX_NONE,\
            OPC_STAT_LOCAL);
rcvd_rm_cell_act = op_stat_reg("RM Cell Activity (Rcvd)",\
            OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
pk_count_stat = op_stat_reg("Packet Count",OPC_STAT_INDEX_NONE,\
            OPC_STAT_LOCAL);

/* Determine the number of fixed input nodes. */
num_fixed_nodes =
op_topo_assoc_count(own_id,OPC_TOPO_ASSOC_IN,OPC_OBJTYPE_QUEUE);

/* Loop through all the input nodes, extracting the value of the */
/* condition attribute and printing value. */
for (i = 0; i < num_fixed_nodes; i++)
    {
    /* Initialize 'port' state variables. */
    accept_grant[i]    = 0;
    accept_request[i]  = 0;
```

```
    /* Each port initially sends a grant. This is in accordance with*/
    /* the example of figure (2) vice the sendFirstGrant protocol */
    /* explained in section IV.B. of the paper by Gauthier et al. */
    sent_grant[i]       = 1;
    sent_request[i]     = 0;
    /* Each port initially sends a grant => sent_seq_num = 1. */
    sent_seq_num[i]     = 1;
    recvd_grant[i]      = 0;
    recvd_request[i]    = 0;
    recvd_seq_num[i]    = 0;
    /* Initially assume all ports to be active. */
    status[i]           = ACTIVE;

    /* Obtain object ID of Nth fixed node. */
    fn_objid = op_topo_assoc (own_id,OPC_TOPO_ASSOC_IN,\
                        OPC_OBJTYPE_QUEUE,i);

    /* Extract value of condition attribute. */
    op_ima_obj_attr_get (fn_objid, "SMART_value", &SMART_neighbor);

    /* Set switch bias for each port. */
    if (SMART_value > SMART_neighbor)
         {
         bias[i] = CANCEL;
         }
    else
         if (SMART_value < SMART_neighbor)
              {
              bias[i] = !CANCEL;
              }
    }

/* Schedule initial RM cells generation for link initialization. */
op_intrpt_schedule_self(op_sim_time(), RM_flag_self);
```

5. Idle State

```
data_request = 0;
```

6. Arrival State

```
/* Acquire stream id number. */
stream_id = op_intrpt_strm();

/* Reset data_request & RM_cell_rcvd. */
data_request = 0;
RM_cell_rcvd = 0;

/* Acquire the arriving cell. */
/* Multiple arriving streams are supported. */
pkptr_1 = op_pk_get (stream_id);
```

```
/* Get payload type (PT) and determine if RM cell. */
op_pk_nfd_get(pkptr_1,"header fields",&atm_hdr_ptr);

/* Read payload type. */
payload_type = atm_hdr_ptr->PT;

/* Reload the ATM header. */
op_pk_nfd_set(pkptr_1,"header
fields",atm_hdr_ptr,op_prg_mem_copy_create,\
               op_prg_mem_free,sizeof(AtmT_Cell_Header_Fields));


if (payload_type!=RM_flag)
     /* This is a DATA cell. */
     {

     /* If the source of this cell is external, then send to sink. */
     if(stream_id==0)
          {
          op_pk_send_forced (pkptr_1, 1);
          pk_count++;
          op_stat_write(pk_count_stat,pk_count);
          insert_ok = 0;
          }

      /* Otherwise, process the data cell. */
     else
          {

          /* Increment total number of cells generated */
          cell_total = cell_total + 1;

          /* If the grant is not held, then generate request to */
          /* transmit data. */
          if(sent_request[0]==0 && accept_grant[0]==0)
               {
               sent_request[0] = 1;
               data_request = 1;
               }

          /* Attempt to enqueue the packet at tail of subqueue. */
          if(cell_queue>=QUEUE_SIZE)
               {

             /* Increment total number of cells destroyed. */
             cell_destroy = cell_destroy + 1;

             /* The insertion failed (due to a full queue). */
             /* Deallocate the packet. */
               op_pk_destroy (pkptr_1);

             /* Set flag indicating insertion fail. This flag is */
             /* used to determine transition out of this state. */
             insert_ok = 0;
```

```
            }
        else
            {
            op_subq_pk_insert(0,pkptr_1,OPC_QPOS_TAIL);

            /* Increment total number of cells successfully loaded */
            /*in queue */
            cell_queue = cell_queue + 1;
              /* Increment total number of data cells in queue. */
              data_cell = data_cell + 1;
              op_stat_write(cell_queue_stat,cell_queue);
            /* insertion was successful. */
              insert_ok = 1;
              }

        /* Calculate cell loss probability. */
        cell_loss = (double)cell_destroy/(double)cell_total;

        /* Write stat to registry. */
        op_stat_write(cell_loss_prob,cell_loss);
        }
    }
else
    /* This is an RM cell. */
    {
    RM_cell_rcvd = 1;
    rm_rcvd_cnt++;
    op_stat_write(rcvd_rm_cell_act,rm_rcvd_cnt);
    insert_ok = 0;
    }
```

7. RM_Rcpt State

```
/* Reset RM_flag_immediate. */
flag_RM_immediate = 0;


/* Update state variables. */
if(RM_cell_rcvd==1)
    {
    /* If due to received RM cell, then read RM cell information. */
    op_pk_nfd_get(pkptr_1,"SG",&recvd_grant[0]);
    op_pk_nfd_get(pkptr_1,"SR",&recvd_request[0]);
    op_pk_nfd_get(pkptr_1,"SSN",&recvd_seq_num[0]);
    }


/* Send First Grant */
/* This is not necessary since the links have already been */
/* initalized. */


/* Cancel First Grant */
```

```
if((bias[0] == CANCEL) && (status[0] == ACTIVE) &&\
   (sent_grant[0] == 1) && (recvd_grant[0] == 1) &&\
   (sent_seq_num[0] == recvd_seq_num[0]))
     {
     sent_grant[0] = 0;
     accept_grant[0] = 1;

     /* A new RM cell must be sent immediately from this port.*/
     flag_RM_immediate = 1;
     }


/* Accept New Request */
if((accept_request[0]==0) && (recvd_request[0]==1))
     {
     accept_request[0] = 1;
     }
/* This does not require that a new RM cell be immediately sent. */


/* Send New Request */
/* This is located in the Arrival Enter Execs. */


/* Accept New Grant */
ssn_flag = sent_seq_num[0] + 1;      /* Modulo 3 */
ssn_flag %= 3;                       /* Addition */
if((accept_grant[0]==0) &&\
   (status[0] ==  ACTIVE) && (recvd_grant[0] == 1) &&\
   (recvd_seq_num[0] == ssn_flag))
     {
     accept_grant[0] = 1;
     sent_grant[0] = 0;
     accept_request[0] = recvd_request[0];
     sent_seq_num[0] = recvd_seq_num[0];
     grant_hold = 1;
     op_stat_write(grant_hold_stat,grant_hold);

     /* If a grant has been accepted, then it is not necessary to */
     /* request to transmit data until the grant has been passed. */
     sent_request[0] = 0;
     flag_RM_immediate = 1;

     /* Reset timer_expired and set timer interrupt. */
     timer_expired = 0;
     evh_2 = op_intrpt_schedule_self(op_sim_time() + \
                         GRANT_time, GRANT_flag);
     }
     /* A new RM cell must be immediately sent from this port. */


/* Send New Grant */
if((status[0]==ACTIVE) && (sent_grant[0]==0) &&\
   (accept_request[0]==1) && (timer_expired==1))
```

```
    {
    sent_grant[0] = 1;
    accept_grant[0] = 0;
    sent_seq_num[0]++;          /* Modulo 3 */
    sent_seq_num[0] %= 3;       /* Addition */
    flag_RM_immediate = 1;
    grant_hold = 0;
    op_stat_write(grant_hold_stat,grant_hold);

    /* If the queue is not empty, then a request to transmit data */
    /* must be sent.*/
    if(data_cell>=1)
        {
        sent_request[0] = 1;
        }
    }
/* A new RM cell must be immediately sent from this port. */


/* Cancel Sent Request */
/* This is found in the "QUE_chk" state. */


/* Destroy received RM cell. */
if(RM_cell_rcvd==1)
    {
    op_pk_destroy (pkptr_1);
    }
```

8. Timer State

```
/* Set timer_expired so that grant may now be released. */
timer_expired = 1;

/* Reset RM_cell_rcvd since this is due to a self interrupt, vice an */
/* RM cell. */
RM_cell_rcvd = 0;
data_request = 0;
insert_ok = 0;
```

9. RM_Gen State

```
/* Schedule next periodic RM cell generation. */
op_intrpt_schedule_self(op_sim_time() + rm_cell_gen_time,
RM_flag_self);
```

10. Que_Chk State

```
update = 0;

/* If the queue is still empty, then reset sent_request and send */
```

```
/* an RM cell. */
if(data_cell==0)
     {
     flag_RM_immediate = 1;
     sent_request[0] = 0;
     update = 1;

     /* Send New Grant */
     if(status[0]==ACTIVE && sent_grant[0]==0 &&\
      accept_request[0]==1)
          {
          sent_grant[0] = 1;
          accept_grant[0] = 0;
          sent_seq_num[0]++;          /* Modulo 3 */
          sent_seq_num[0] %= 3;       /* Addition */
          flag_RM_immediate = 1;

          /* Clear any pending timer interrupts. */
          if(op_ev_valid(evh_2)==1)
               {
               op_ev_cancel(evh_2);
               }
          }
     /* A new RM cell must be immediately sent from this port. */
     }
```

11. RM_Xmit State

```
/* This process only interfaces with one port: source stream zero. */

/* Create RM cell. */
cell_ptr = op_pk_create_fmt("rrw_ams_atm_rm");

/* Allocate memory for the header and assign RM field. */
atm_hdr_ptr = set_header(1);

/* Load the ATM header. */
op_pk_nfd_set(cell_ptr,"header
fields",atm_hdr_ptr,op_prg_mem_copy_create,\
              op_prg_mem_free,sizeof(AtmT_Cell_Header_Fields));

/* Load state variable values. */
op_pk_nfd_set(cell_ptr,"SG",sent_grant[0]);
op_pk_nfd_set(cell_ptr,"SR",sent_request[0]);
op_pk_nfd_set(cell_ptr,"SSN",sent_seq_num[0]);

/* Load metric based on current queue size and current delay time. */
metric = delay_metric*data_cell;
op_pk_nfd_set(cell_ptr,"Queue_Size",metric);

if(sent_request[0]==0)
     {
     source_id = 0;
```

```
     }
else
     {
     source_id = temp;
     }
op_pk_nfd_set(cell_ptr,"ID",source_id);

/* Clear dummy RM cell flag. */
op_pk_nfd_set(cell_ptr,"Rsvd",0);

/* Transmit RM cell on source stream zero. */
op_pk_send(cell_ptr,0);
```

12. Svc_Start State

```
/* Schedule an interrupt for this process at the time where */
/* service ends. */
op_intrpt_schedule_self (op_sim_time () + pk_svc_time, CELL_flag_xmit);

/* the server is now busy. */
server_busy = 1;

/* Reset data_request. */
data_request = 0;
```

13. Svc_Compl State

```
/* Get source ID. */
op_ima_obj_attr_get (own_id, "source_id", &source_id);

/* Extract packet at head of queue. */
/* This is the packet just finishing service. */
pkptr_1 = op_subq_pk_remove (0, OPC_QPOS_HEAD);

/* This is a data cell */
if(accept_grant[0]==1)
     {

     /* This algorithm is switch specific and will change if the */
     /* node configuration changes. SW1 = 14; SW2 = 17; SW3 = 25. */
     switch(source_id)
          {
          case 1:
               status_ptr = (int *) op_ima_obj_svar_get(25,"status");
               status_ptr[2] = INACTIVE;

               status_ptr = (int *) op_ima_obj_svar_get(17,"status");
               status_ptr[3] = INACTIVE;
               break;

          case 2:
               status_ptr = (int *) op_ima_obj_svar_get(14,"status");
```

```
                status_ptr[3] = INACTIVE;

                status_ptr = (int *) op_ima_obj_svar_get(25,"status");
                status_ptr[1] = INACTIVE;
                break;

            case 3:

                status_ptr = (int *) op_ima_obj_svar_get(14,"status");
                status_ptr[2] = INACTIVE;

                status_ptr = (int *) op_ima_obj_svar_get(17,"status");
                status_ptr[2] = INACTIVE;
                break;
            }

        /* Decrement total cell count in queue. */
        cell_queue = cell_queue - 1;

        /* Decrement total number of data cells in queue. */
        data_cell = data_cell -1;
        op_stat_write(cell_queue_stat,cell_queue);
        op_pk_send_forced (pkptr_1, 0);
        server_busy = 0;
        }

    /* Otherwise, check the data cell's age.  If the data cell is too */
    /* old, then discard it.  If not, then this data cell should not */
    /* be de-queued yet.  Return it to its original position at the */
    /* head of the subqueue. */
    else
        {

        /* Get time of origin and compute delay. */
        op_pk_nfd_get(pkptr_1,"Time_Start",&Time_Start);
        delay = op_sim_time() - Time_Start;


        /* Set delay metric. */
        switch(source_id)
            {
            case 1: case 2:
                if(delay<=.010)
                    {
                    delay_metric = 1.0;
                    }
                else if(delay<=.020)
                    {
                    delay_metric = 2.72;
                    }
                else if(delay<=.030)
                    {
                    delay_metric = 7.39;
                    }
```

```
        else if(delay<=.040)
             {
             delay_metric = 20.09;
             }
        else if(delay<=.050)
             {
             delay_metric = 54.60;
             }
        else if(delay<=.060)
             {
             delay_metric = 148.41;
             }
        else if(delay<=.070)
             {
             delay_metric = 403.43;
             }
        else if(delay<=.080)
             {
             delay_metric = 1096.63;
             }
        else if(delay<=.090)
             {
             delay_metric = 2980.96;
             }
        else if(delay<=.100)
             {
             delay_metric = 8103.08;
             }
        else if(delay<=.110)
             {
             delay_metric = 22026.47;
             }
        else if(delay<=.120)
             {
             delay_metric = 59874.0;
             }
        else if(delay<=.130)
             {
             delay_metric = 162754.0;
             }
        else if(delay<=.140)
             {
             delay_metric = 442413.0;
             }
        else
             {
             delay_metric = 1202604.0;
             }
        break;
   case 3:
        if(delay<=.010)
             {
             delay_metric = 1.0*boost;
             }
```

```
else if(delay<=.020)
      {
      delay_metric = 2.72*boost;
      }
else if(delay<=.030)
      {
      delay_metric = 7.39*boost;
      }
else if(delay<=.040)
      {
      delay_metric = 20.09*boost;
      }
else if(delay<=.050)
      {
      delay_metric = 54.60*boost;
      }
else if(delay<=.060)
      {
      delay_metric = 148.41*boost;
      }
else if(delay<=.070)
      {
      delay_metric = 403.43*boost;
      }
else if(delay<=.080)
      {
      delay_metric = 1096.63*boost;
      }
else if(delay<=.090)
      {
      delay_metric = 2980.96*boost;
      }
else if(delay<=.100)
      {
      delay_metric = 8103.08*boost;
      }
else if(delay<=.110)
      {
      delay_metric = 22026.47*boost;
      }
else if(delay<=.120)
      {
      delay_metric = 59874.0*boost;
      }
else if(delay<=.130)
      {
      delay_metric = 162754.0*boost;
      }
else if(delay<=.140)
      {
      delay_metric = 442413.0*boost;
      }
else
      {
```

```
                    delay_metric = 1202604.0*boost;
                    }
                break;
            }
        /* Is this cell older than MAX_DELAY?  If not, then requeue. */
        if(delay<=(MAX_DELAY - SOURCE_time))
            {
            op_subq_pk_insert(0,pkptr_1,OPC_QPOS_HEAD);
            }
        /* Otherwise, discard. */
        else
            {
            /* Increment total number of cells destroyed. */
          cell_destroy = cell_destroy + 1;

            /* Decrement total cell count in queue. */
            cell_queue = cell_queue - 1;

            /* Decrement total number of data cells in queue. */
            data_cell = data_cell -1;
            op_stat_write(cell_queue_stat,cell_queue);

          /* Deallocate the packet. */
            op_pk_destroy (pkptr_1);

            /* Calculate cell loss probability. */
            cell_loss = (double)cell_destroy/(double)cell_total;

            /* Write stat to registry. */
            op_stat_write(cell_loss_prob,cell_loss);
            }

        /* Server is idle again. */
        server_busy = 0;
        }

/* If the queue is empty of data cells, then schedule interrupt */
/* to clear sent_request if the queue is still empty after the */
/* inverse of the service rate. */
if(data_cell==0)
        {

        /* Clear previous Queue Check interrupts (if any). */
        if(op_ev_valid(evh_1)==1)
            {
            op_ev_cancel(evh_1);
            }

        /* Schedule next interrupt. */
        evh_1 = op_intrpt_schedule_self(op_sim_time() + .000001,
Queue_Check);
        }

/* Prevent transition to svc_start via RM_xmit. */
```

```
data_request = 0;
```

F. SMART SWITCH MODEL

The OPNET model for the SMART switch discussed in chapters IV and V is shown in Figure A-6. All code shown for all states except for the idle state is located in the "Enter Execs" block of each state. Code for the idle state is located in the "Exit Execs" block for that state.

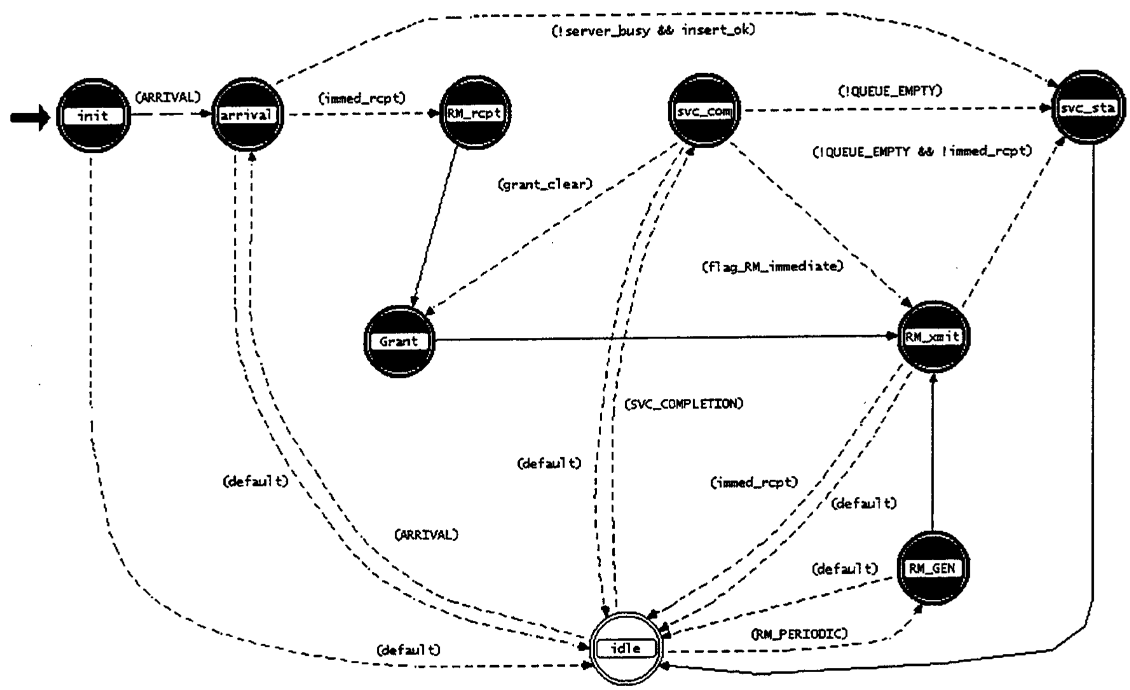

Figure A-6 Finite State Machine for a SMART Switch

1. Header Block

```
OPC_COMPILE_CPP

#include "iostream.h"
#include "ams_pk_support.h"

/* Define constants. */
#define GRANT_time      .010
#define RM_mult         13
#define RM_flag         6
```

```
#define QUEUE_SIZE      100
#define ACTIVE          1
#define INACTIVE  0
#define CANCEL          1

/* Set 'port' array size */
/* Determined by number of input/ouput ports. */
/* Port_Sz is 3 for smart_sw3. */
/* Port_Sz is 4 for smart_sw1 and smart_sw2. */
#define Port_Sz         3

/* Define op codes for self interrupts. */
#define CELL_flag_xmit  0
#define RM_flag_recd    1
#define GRANT_flag      2
#define RM_flag_self    3

/* Define macros for transition states. */
#define QUEUE_EMPTY     (op_q_empty ())

#define ARRIVAL         op_intrpt_type () == OPC_INTRPT_STRM
#define SVC_COMPLETION  (op_intrpt_code () == CELL_flag_xmit) &&\
                  (op_intrpt_type () == OPC_INTRPT_SELF)
#define RM_PERIODIC     (op_intrpt_code () == RM_flag_self) &&\
                  (op_intrpt_type () == OPC_INTRPT_SELF)
#define TIME_EXPIRED    (op_intrpt_code () == GRANT_flag) &&\
                  (op_intrpt_type () == OPC_INTRPT_SELF)

/* Define struct function. */
AtmT_Cell_Header_Fields* set_header(int);

AtmT_Cell_Header_Fields* atm_hdr_ptr;
```

2. State Variables Block

```
int    \fn_objid;
int    \num_fixed_nodes;
int    \SMART_neighbor;
int    \SMART_value;
int    \flag_RM_immediate;
int    \server_busy;
int    \RM_dummy;
int    \cell_total;
int    \cell_destroy;
int    \cell_queue;
int    \rm_rcvd_cnt;
int    \accept_grant[Port_Sz];
int    \accept_request[Port_Sz];
int    \sent_grant[Port_Sz];
int    \sent_request[Port_Sz];
int    \sent_seq_num[Port_Sz];
int    \recvd_grant[Port_Sz];
int    \recvd_request[Port_Sz];
```

```
int    \recvd_seq_num[Port_Sz];
int    \bias[Port_Sz];
int    \status[Port_Sz];
int    \RM_port_flag[Port_Sz];
int    \insert_ok;
int    \port_priority;
int    \current_port;
int    \grant_flag[Port_Sz];
int    \source_id[Port_Sz];
int    \switch_id;
int    \stream_id;
int    \past_port;
int    \num_fixed_nodes_j;
int    \j;
int    \link_objid;
int    \link;
int    \q;
int    \data_cell;
int    \grant_clear;
int    \grant_hold;
int    \out_stream;
int    \request_flag;
int    \xmit_hold[Port_Sz];
int    \canx_flag;
int    \immed_rcpt;
int    \link_hold;
int    \queue_size[Port_Sz];

int * \status_ptr;

double       \port_timer[Port_Sz];
double       \rm_cell_gen_time;
double       \rm_cell_sched_time;
double       \service_rate;
double       \cell_loss;
double       \pk_svc_time;
double       \Time_Stamp;

Packet *     \pkptr_1;

Objid \own_id;

Stathandle  \cell_loss_prob;
Stathandle  \cell_queue_stat;
Stathandle  \rcvd_rm_cell_act;
```

3. Temporary Variables Block

```
int                     i;
int                     tie_flag;
int                     max_queue;
int                     payload_type;
int                     accept_flag;
```

```
int                     clear_flag;
int                     request_flag;
int                     groom_flag;
int                     send_flag;
int                     ssn_flag;
int                     canx_flag;
int                     test_ptr;
int                     rm_flag;
int                     node;

double                        pk_len;
double                        min_time;

AtmT_Cell_Header_Fields*      atm_hdr_ptr;

Packet *                pkptr_2;
Packet *                pkptr_copy;
Packet *                cell_ptr;
```

4. Function Block

```
AtmT_Cell_Header_Fields* set_header(int rm_flag)
{
     AtmT_Cell_Header_Fields* atm_hdr_ptr;

     /* Allocate memory for header fields. */
     atm_hdr_ptr = (AtmT_Cell_Header_Fields*)op_prg_mem_alloc(\
                                   sizeof(AtmT_Cell_Header_Fields));

     /* Load the payload type (RM cell = 110 binary [6 decimal]). */
     if(rm_flag == 1)
          {
          atm_hdr_ptr->PT = RM_flag;
          }
     else
          {
          atm_hdr_ptr->PT = 0;
          }
     return atm_hdr_ptr;
}
```

5. Init State

```
/* initially the server is idle */
server_busy = 0;
rm_rcvd_cnt = 0;

/* Get queue module's own object id. */
own_id = op_id_self ();

/* Get assigned value of server processing rate. */
op_ima_obj_attr_get (own_id, "service_rate", &service_rate);
```

```
/* Get switch ID. */
op_ima_obj_attr_get (own_id, "switch_id", &switch_id);

/* Set dummy SMART value. */
op_ima_obj_attr_get (own_id, "SMART_value", &SMART_value);

pk_svc_time = 1.0/service_rate;
rm_cell_gen_time = RM_mult*pk_svc_time;

/* Initialize state variables. */
link_hold          = 0;
grant_clear        = 0;
grant_hold         = 0;
data_cell          = 0;
cell_total         = 0;
cell_destroy       = 0;
cell_queue         = 0;
cell_loss          = 0;
cell_loss_prob     = op_stat_reg("Cell Loss Probability",\
                   OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
cell_queue_stat = op_stat_reg("Queue Size Stat",\
                   OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
rcvd_rm_cell_act = op_stat_reg("RM Cell Activity (Rcvd)",\
                   OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);

/* Determine the number of fixed input nodes. */
num_fixed_nodes =
op_topo_assoc_count(own_id,OPC_TOPO_ASSOC_IN,OPC_OBJTYPE_QUEUE);

/* Loop through all the input nodes, extracting the value of the
condition */
/* attribute and printing value. */
for (i = 0; i < num_fixed_nodes; i++)
     {

     /* Initialize 'port' state variables. */
     queue_size[i]       = 0;
     xmit_hold[i] = 0;
     accept_grant[i]     = 0;
     accept_request[i]   = 0;

     /* Each port initially sends a grant. This is in accordance */
     /* with the example of figure (2) vice the sendFirstGrant */
     /* protocol explained in section IV.B. of the paper by Gauthier */
     /* et al. */
     sent_grant[i]       = 1;
     sent_request[i]     = 0;

     /* Each port initially sends a grant => sent_seq_num = 1. */
     sent_seq_num[i]     = 1;
     recvd_grant[i]      = 0;
     recvd_request[i]    = 0;
     recvd_seq_num[i]    = 0;
```

```
        /* Source ID of zero indicates RM cell originates from */
        /* switch, vice a workstation. */
        source_id[i] = 0;

        /* Initially assume all ports to be active. */
        status[i]          = ACTIVE;

        /* Initialize flag variables. */
        RM_port_flag[i]    = 0;
        port_timer[i]      = op_sim_time();

        /* Activate the grant_flag status for all ports. */
        grant_flag[i]      = 1;

        /* Obtain object ID of Nth fixed node. */
        fn_objid = op_topo_assoc (own_id,OPC_TOPO_ASSOC_IN,\
                                  OPC_OBJTYPE_QUEUE,i);

        /* Extract value of condition attribute. */
        op_ima_obj_attr_get (fn_objid, "SMART_value", &SMART_neighbor);

        /* Set switch bias for each port. */
        if (SMART_value > SMART_neighbor)
            {
            bias[i] = CANCEL;
            }
        else
            if (SMART_value < SMART_neighbor)
                {
                bias[i] = !CANCEL;
                }
        }

/* Schedule initial RM cells generation for link initialization. */
op_intrpt_schedule_self(op_sim_time(), RM_flag_self);
```

6. Arrival State

```
/* Reset RM_flag_immediate. */
flag_RM_immediate = 0;

/* Acquire stream id number. */
stream_id = op_intrpt_strm();

/* Acquire the arriving packet. Multiple arriving streams are */
/* supported. */
pkptr_1 = op_pk_get (stream_id);

/* Get payload type (PT) and determine if RM cell. */
op_pk_nfd_get(pkptr_1,"header fields",&atm_hdr_ptr);

/* Read payload type. */
payload_type = atm_hdr_ptr->PT;
```

```
/* Reload the ATM header. */
op_pk_nfd_set(pkptr_1,"header
fields",atm_hdr_ptr,op_prg_mem_copy_create,\
              op_prg_mem_free,sizeof(AtmT_Cell_Header_Fields));

if (payload_type!=RM_flag)
     /* This is a data cell */
     {

     /* Clear the link hold which may have been generated by the */
     /* "Grant" state. */
     link_hold = 0;

     /* Increment total number of cells generated */
     cell_total = cell_total + 1;

     /* Attempt to enqueue the packet at tail of subqueue. */
     if(cell_queue>=QUEUE_SIZE)

          {
        /* Increment total number of cells destroyed. */
        cell_destroy = cell_destroy + 1;

        /* The insertion failed (due to a full queue). */
        /* Deallocate the packet. */
          op_pk_destroy (pkptr_1);

        /* Set flag indicating insertion fail. This flag is used to */
        /* determine transition out of this state. */
        insert_ok = 0;
          }
     else {
        op_subq_pk_insert(0,pkptr_1,OPC_QPOS_TAIL);

        /* Increment total number of cells successfully loaded in */
        /* queue. */
        cell_queue = cell_queue + 1;
        op_stat_write(cell_queue_stat,cell_queue);

        /* Increment total number of data cells in queue. */
        data_cell = data_cell+1;

        /* Insertion was successful. */
          insert_ok = 1;
          }

     /* Calculate cell loss probability. */
     cell_loss = (double)cell_destroy/(double)cell_total;

     /* Write stat to registry. */
     op_stat_write(cell_loss_prob,cell_loss);

     /* Clear flag which causes immediate transition to RM_rcpt. */
```

```
      immed_rcpt = 0;
      }
else
      /* This is an RM cell. */
      {
      /* Insert stream ID value into RM cell. */
      op_pk_nfd_set(pkptr_1,"Rsvd_ID",stream_id);

      /* This RM cell is not in the data stream and will not be */
      /* queued. */
      immed_rcpt = 1;
      insert_ok = 0;

      rm_rcvd_cnt++;
      op_stat_write(rcvd_rm_cell_act,rm_rcvd_cnt);
      }
```

7. RM_Rcpt State

```
/* Check whether this is an externally or internally generated RM */
/* cell. */
op_pk_nfd_get(pkptr_1,"Rsvd",&RM_dummy);

/* If external RM cell, then update state variables. */
if(RM_dummy == 0)
      {

      /* Obtain original stream id from RM cell. */
      op_pk_nfd_get(pkptr_1,"Rsvd_ID",&stream_id);
      if(status[stream_id]==ACTIVE)
            {

            /* Read RM cell information. */
            op_pk_nfd_get(pkptr_1,"SG",&recvd_grant[stream_id]);
            op_pk_nfd_get(pkptr_1,"SSN",&recvd_seq_num[stream_id]);
            op_pk_nfd_get(pkptr_1,"ID",&source_id[stream_id]);
            }

      op_pk_nfd_get(pkptr_1,"SR",&recvd_request[stream_id]);

      /* Update Queue Size Data */
      /* The queue_size must be updated each time a request to send */
      /* data is received from a switch or each time that an RM cell */
      /* is received from a work station. */
      if((switch_id==1 || switch_id==2) && (stream_id==0 || \
         stream_id==1) || switch_id==3 && stream_id==0 || \
         recvd_request[stream_id]==1)
            {
            op_pk_nfd_get(pkptr_1,"Queue_Size",&queue_size[stream_id]);
            }


      /* Cancel Accepted Request */
```

```
    if(recvd_request[stream_id]==0 || \
            status[stream_id]==INACTIVE)
          {
          accept_request[stream_id] = 0;
          }


/* Send First Grant */
/* This is not necessary since the links have already been */
/* initalized. */


/* Cancel First Grant */
canx_flag = 1;
for(i=0;i<num_fixed_nodes;i++)
    {
    if((status[i]==ACTIVE) && (i!=stream_id) && \
                                (op_sim_time()>2))
          {
          canx_flag = 0;
          }
    }

if((bias[stream_id] == CANCEL) && (status[stream_id] == ACTIVE) \
 && (sent_grant[stream_id] == 1) && (recvd_grant[stream_id] == 1)\
 && (sent_seq_num[stream_id] == recvd_seq_num[stream_id]) &&\
    (canx_flag==1))
    {
    sent_grant[stream_id] = 0;
    accept_grant[stream_id] = 1;

    /* A new RM cell must be sent immediately from this port.*/
    RM_port_flag[stream_id] = 1;
    flag_RM_immediate = 1;
    }

/* Accept New Request */
/* Data request RM cells are sent via "ACTIVE" and "INACTIVE" */
/* ports.  However, data requests are only accepted via */
/* "ACTIVE" ports. */
if((accept_request[stream_id]==0)&&(recvd_request[stream_id]==1)\
      &&(status[stream_id]==ACTIVE))
    {

    /* If an outstanding data request from another node */
    /* exists, do not reroute paths. */
    request_flag = 1;
    for(i=0;i<num_fixed_nodes;i++)
          {
          if(accept_request[i]==1)
                {
                request_flag = 0;
                }
          }
```

```
accept_request[stream_id] = 1;

/* If an outstanding data request from another node does */
/* not exist, then reroute paths.*/
if(request_flag==1 && link_hold==0)
     {
     /* This is intended to prevent parallel paths for RM */
     /* cells requesting for a source to transmit data. */
     switch(switch_id)
          {
          case 1:
               if((source_id[stream_id]==1) && \
              (status[2]==ACTIVE) && (status[3]==ACTIVE))
                    {
                    status_ptr = (int *) op_ima_obj_svar_get\
                                        (17,"status");
                    status_ptr[3] = INACTIVE;
                    status_ptr = (int *) op_ima_obj_svar_get\
                                        (25,"status");
                    status_ptr[2] = INACTIVE;
                    }
               break;

          case 2:
               if((source_id[stream_id]==2) && \
                    (status[2]==ACTIVE) && \
                    (status[3]==ACTIVE))
                    {
                    status_ptr = (int *) op_ima_obj_svar_get\
                                        (14,"status");
                    status_ptr[3] = INACTIVE;
                    status_ptr = (int *) op_ima_obj_svar_get\
                                        (25,"status");
                    status_ptr[1] = INACTIVE;
                    }
               break;

          case 3:
               if((source_id[stream_id]==3) && \
                    (status[1]==ACTIVE) && \
                    (status[2]==ACTIVE))
                    {
                    status_ptr = (int *) op_ima_obj_svar_get\
                                        (14,"status");
                    status_ptr[2] = INACTIVE;
                    status_ptr = (int *) op_ima_obj_svar_get\
                                        (17,"status");
                    status_ptr[2] = INACTIVE;
                    }
               break;
          }
     }
  }
/* This does not require that a new RM cell be immediately */
```

```
/* sent. */


/* Send New Request */
if(accept_request[stream_id]==1 && status[stream_id]==ACTIVE)
     {
     for(i=0;i<num_fixed_nodes;i++)
          {
          if(i!=stream_id && sent_request[i]==0)
               {
               RM_port_flag[i] = 1;
               }
          }
     flag_RM_immediate = 1;

     /* A request should only be sent via a port other than */
     /* the port that has accepted a request. Also, requests */
     /* should only be sent from a workstation to a switch */
     /* (via a switch) or from a switch to a workstation (via */
     /* a switch). Requests sent from a switch (via a switch) */
     /* to another switch (i.e. sw1 to sw3 to sw2) will result */
     /* in conditions which are not allowed. */
     switch(switch_id)
          {
          case 1: case 2:
               if((stream_id==0 || stream_id==1))
                    {
                    sent_request[2] = 1;
                    sent_request[3] = 1;
                    if(accept_request[0]==0 && \
                         accept_request[1]==1)
                         {
                         sent_request[0] = 1;
                         sent_request[1] = 0;
                         }
                    else if(accept_request[0]==1 && \
                         accept_request[1]==0)
                         {
                         sent_request[0] = 0;
                         sent_request[1] = 1;
                         }
                    else
                         {
                         sent_request[0] = 1;
                         sent_request[1] = 1;
                         }
                    }
               else if(stream_id==2 || stream_id==3)
                    {
                    sent_request[0] = 1;
                    sent_request[1] = 1;
                    if(accept_request[0]==1 || \
                         accept_request[1]==1)
                         {
```

```
                        sent_request[2] = 1;
                        sent_request[3] = 1;
                        }
                  else
                        {
                        sent_request[2] = 0;
                        sent_request[3] = 0;
                        }
                  }
            break;
        case 3:
            if(stream_id==0)
                  {
                  sent_request[1] = 1;
                  sent_request[2] = 1;
                  }
            else if(stream_id==1 || stream_id==2)
                  {
                  sent_request[0] = 1;
                  if(accept_request[0]==1)
                        {
                        sent_request[1] = 1;
                        sent_request[2] = 1;
                        }
                  else
                        {
                        sent_request[1] = 0;
                        sent_request[2] = 0;
                        }
                  }
            break;
        }

    }

/* Update Port Timer */
/* If this data request is from another switch, then update */
/* the port timer. It is required that port timer updates pass */
/* through both "ACTIVE" and "INACTIVE" links to ensure that */
/* ports remain updated. */
if(((switch_id==1 || switch_id==2) && (stream_id==2 || \
       stream_id==3) || switch_id==3 && (stream_id==1 || \
       stream_id==2)) && accept_request[stream_id]==1)
     {
     op_pk_nfd_get(pkptr_1,"Time_Stamp",&Time_Stamp);
     port_timer[stream_id] = Time_Stamp;
     }

/* Accept New Grant */
ssn_flag = sent_seq_num[stream_id] + 1;     /* Modulo 3 */
ssn_flag %= 3;                              /* Addition */
if(accept_grant[stream_id]==0 &&\
 status[stream_id] ==  ACTIVE && recvd_grant[stream_id]==1 &&\
 recvd_seq_num[stream_id] == ssn_flag)
```

```
      {
      accept_grant[stream_id] = 1;
      sent_grant[stream_id] = 0;
      accept_request[stream_id] = recvd_request[stream_id];
      sent_seq_num[stream_id] = recvd_seq_num[stream_id];
      xmit_hold[stream_id] = 1;
      }
/* This does not require that a new RM cell be immediately */
/* sent. */


/* Send New Grant */
send_flag = 1;
request_flag = 0;
for(i=0;i<num_fixed_nodes;i++)
      {
      if(status[i]==ACTIVE && (accept_grant[i]==0 || \
                     sent_grant[i]==1))
            {
            send_flag = 0;
            }
      if(status[i]==ACTIVE && accept_request[i]==1)
            {
            request_flag = 1;
            }
      }
if(request_flag==1 && send_flag==1 && grant_hold==0)

      {
      /* If the conditions for a grant are met and the queue is */
      /* empty of data cells, then clear a grant to be */
      /* transmitted. */
      if(data_cell==0)
            {
            grant_clear = 1;
            }

      /* If the conditions for a grant are met and the queue is */
      /* not empty of data cells, then hold the grant until the */
      /* queue is ready. */
      else
            {
            grant_hold = 1;
            }
      }


/* Groom Sent Request */
/* In order to prevent an endless loop between two switches, it */
/* is necessary to ensure that a switch does not receive a sent */
/* request from a switch that it sends a sent_request to */
/* (unless the other switch has received a sent_request from */
/* a third source). */
```

```
	/* See svc_comp for "Cancel Sent Request" function. */
	groom_flag = 1;
	for(i=0;i<num_fixed_nodes;i++)
		{
		if(accept_request[i]==1 && status[i]==ACTIVE && i!=stream_id)
			{
			groom_flag = 0;
			}
		}
	if(groom_flag==1 && accept_request[stream_id]==1)
		{
		sent_request[stream_id] = 0;
		}
	}

/* Destroy received RM cell. */
op_pk_destroy (pkptr_1);
```

8. Grant State

```
/* If a grant has been scheduled and the queue is empty of data */
/* cells, then pass the grant. */
if(grant_clear==1)
	{
	grant_clear = 0;

	/* Find the port with an outstanding request to transmit */
	/* data which has transmitted least recently or has the */
	/* largest queue occupancy. */
	min_time = op_sim_time();
	max_queue = 0;
	tie_flag = 0;
	for(i=0;i<num_fixed_nodes;i++)
		{
		/* Clear transmit hold on all ports. */
		xmit_hold[i] = 0;

		/* Port which has transmitted least recently. */
		if(port_timer[i]<=min_time && recvd_request[i]==1 \
				&& status[i]==ACTIVE)
			{
			min_time = port_timer[i];
			}

		/* Port corresponding to largest queue occupancy. */
		if(queue_size[i]>=max_queue && accept_request[i]==1 \
				&& status[i]==ACTIVE)
			{
			if(queue_size[i]==max_queue && max_queue!=0)
				{
				tie_flag++;
				}
			max_queue = queue_size[i];
```

```
            }
        }
 for(i=0;i<num_fixed_nodes;i++)
        {
        /* If it is desired to give preference in passing the */
        /* grant based on work station queue size then remark */
        /* out the first "if" statement, otherwise remark out */
        /* second "if" statement and the following "switch" */
        /* statement. */
        if(recvd_request[i]==1 && status[i]==ACTIVE \
                    && queue_size[i]==max_queue)
             {
             node = i;
             }
        }

/* If at least two queues have the same non-zero occupancy */
/* levels, then choose the port that corresponds to an */
/* adjacent workstation (vice a switch). This ensures */
/* that an unallowed condtion does not occur upon grant */
/* transfer. */
if(tie_flag>=1)
     {
     switch(switch_id)
          {
          case 1: case 2:
               if(queue_size[0]==max_queue)
                    {
                    node = 0;
                    }
               else if(queue_size[1]==max_queue)
                    {
                    node = 1;
                    }
               break;
          case 3:
               if(queue_size[0]==max_queue)
                    {
                    node = 0;
                    }
               break;
          }
     }

/* When such a port has been found, then change the current */
/* port and the grant_flag status of all ports. Also check */
/* if port changed or is simply "renewed." */
for(i=0;i<num_fixed_nodes;i++)
     {
     if((sent_grant[i]==1) && (status[i]==ACTIVE))
          {
          past_port = i;
          }
     grant_flag[i] = 0;
```

```
     sent_grant[i] = 0;
     }
current_port = node;
grant_flag[current_port] = 1;
sent_grant[current_port] = 1;
accept_grant[current_port] = 0;
sent_seq_num[current_port]++;                              /* Modulo 3 */
sent_seq_num[current_port] %= 3;                           /* Addition */

/* Check for disabled link condition and reactivate link. */
/* It is desired that an inactive link be reactivated only if */
/* the grant is being passed from a smart switch to a */
/* workstation. This occurs (in the delta configuration) */
/* during the process of passing the grant from one workstation */
/* to another via the established link. In this case, */
/* the link which requires reactivation is attached to the */
/* smart switch which is passing the grant to a workstation. */
/* In the same instance, the first smart switch which receives*/
/* the grant (i.e. workstation to smart switch) must deactivate */
/* one of the "outgoing" links in order to properly establish */
/* the routing paths. It is possible to inadvertently */
/* reactivate this link, which is not desirable at this time */
/* and therefore must be prevented. */
if((switch_id==1 || switch_id==2) && (current_port==0 ||/
    current_port==1) || (switch_id==3 && current_port==0))
     {

     /* Once the link has been reactivated, it is possible to */
     /* receive a data request from another workstation prior */
     /* to data transmission from the intended workstation, */
     /* which will incorrectly reconfigure the link. To prevent */
     /* this occurrence, a hold must be placed on the path */
     /* rerouting function in the "Accept New Request" */
     /* transition within the "RM_rcpt" state. */
     link_hold = 1;
     for(i=0;i<num_fixed_nodes;i++)
          {
          if(status[i]==INACTIVE)
               {
               accept_grant[i] = 1;
               sent_grant[i] = 0;
               sent_request[i] = 1;
               recvd_seq_num[i] = 2;
               sent_seq_num[i] = 2;
               status[i] = ACTIVE;

               /* Obtain object ID of associated inactive node. */
               fn_objid = op_topo_assoc(own_id,OPC_TOPO_ASSOC_IN,\
                    OPC_OBJTYPE_QUEUE,i);

               /* Obtain associated link for this node. */
               num_fixed_nodes_j = op_topo_assoc_count(fn_objid,\
                    OPC_TOPO_ASSOC_IN,OPC_OBJTYPE_QUEUE);
               for(j=0;j<num_fixed_nodes_j;j++)
```

```
                {
                link_objid = op_topo_assoc(fn_objid,\
                     OPC_TOPO_ASSOC_IN,OPC_OBJTYPE_QUEUE,j);
                if(link_objid==own_id)
                      {
                      link = j;
                      }
                }

          status_ptr = (int *)op_ima_obj_svar_get(fn_objid,\
                   "sent_request");
          accept_request[i] = status_ptr[link];
          recvd_request[i] = accept_request[i];


          status_ptr = (int *)op_ima_obj_svar_get(fn_objid,\
                   "accept_grant");
          status_ptr[link] = 0;

          status_ptr = (int *)op_ima_obj_svar_get(fn_objid,\
                    "sent_grant");
          status_ptr[link] = 1;

          status_ptr = (int *)op_ima_obj_svar_get(fn_objid,\
                   "accept_request");
          status_ptr[link] = 1;

          status_ptr = (int *)op_ima_obj_svar_get(fn_objid,\
                   "recvd_seq_num");
          status_ptr[link] = 1;

          status_ptr = (int *)op_ima_obj_svar_get(fn_objid,\
                   "sent_seq_num");
          status_ptr[link] = 2;

          status_ptr = (int *)op_ima_obj_svar_get(fn_objid,\
                   "status");
          status_ptr[link] = ACTIVE;

          status_ptr = (int *)op_ima_obj_svar_get(fn_objid,\
                   "recvd_request");
          status_ptr[link] = 1;
          }
     }

/* Set port timer. */
/* If the grant is being sent to a workstation, then the */
/* port timer which corresponds to that workstation must */
/* be updated. */
port_timer[current_port] = op_sim_time();
}

/* A new RM cell must be immediately sent from this port. */
RM_port_flag[current_port] = 1;
```

```
         flag_RM_immediate = 1;

        switch(switch_id)
              {
              case 1: case 2:
                   if(recvd_request[0]==0 && recvd_request[1]==0)
                        {
                        accept_request[0] = 0;
                        accept_request[1] = 0;
                        sent_request[2] = 0;
                        sent_request[3] = 0;
                        }
                   break;
              case 3:
                   if(recvd_request[0]==0)
                        {
                        accept_request[0] = 0;
                        sent_request[1] = 0;
                        sent_request[2] = 0;
                        }
                   break;
              }
    }
```

9. RM_Xmit State

```
/* Examine workstation ports which have a current data request. Of */
/* these ports, determine which has the oldest transmit time or */
/* largest queue occupancy. This time and queue occupancy will be */
/* sent in the outgoing RM cell. */
min_time = op_sim_time();
max_queue = 0;
for(i = 0; i<num_fixed_nodes; i++)
     {

     /* Identify ports that connect directly to workstations. */
     if((switch_id==1 || switch_id==2) && (i==0 || i==1) || \
        (switch_id==3 && i==0))
          {

          /* If this port has an active data request then compare */
          /* the transmit time. */
          if(accept_request[i]==1 && port_timer[i]<=min_time)
               {
               min_time = port_timer[i];
               }
          /* If this port has an active data request then compare */
          /* the queue occupancy. */
          if(accept_request[i]==1 && queue_size[i]>=max_queue)
               {
               max_queue = queue_size[i];
               }
          }
```

```
    }
/* Generate and transmit RM cells to all links if this is a periodic */
/* RM cell. Generate and transmit RM cells to selected links if this */
/* is an immediate RM cell. */
for(i = 0; i<num_fixed_nodes; i++)
    {
    if(RM_dummy==1 || (flag_RM_immediate==1 && RM_port_flag[i]==1))
        {

        /* Create RM cell. */
        cell_ptr = op_pk_create_fmt("rrw_ams_atm_rm");

        /* Allocate memory for the header and assign RM field.  */
        atm_hdr_ptr = set_header(1);

        /* Load the ATM header. */
        op_pk_nfd_set(cell_ptr,"header fields",\
          atm_hdr_ptr,op_prg_mem_copy_create,\
          op_prg_mem_free,sizeof(AtmT_Cell_Header_Fields));

        /* Load state variable values. */
        op_pk_nfd_set(cell_ptr,"SG",sent_grant[i]);
        op_pk_nfd_set(cell_ptr,"SR",sent_request[i]);
        op_pk_nfd_set(cell_ptr,"SSN",sent_seq_num[i]);
        if(sent_request[i]==1)
            {
            op_pk_nfd_set(cell_ptr,"Time_Stamp",min_time);
            op_pk_nfd_set(cell_ptr,"Queue_Size",max_queue);
            }

        /* Clear dummy RM cell flag. */
        op_pk_nfd_set(cell_ptr,"Rsvd",0);

        /* RM cell originates from a SMART switch, therefore */
        /* ID = 0. */
        op_pk_nfd_set(cell_ptr,"ID",0);

        /* Transmit RM cell on appropriate stream. */
        op_pk_send(cell_ptr,i);

        /* Reset RM_port_flag. */
        RM_port_flag[i] = 0;
        }
    }
/* If this was a queued cell, then the server is idle again. */
if(immed_rcpt==0)
    {
    server_busy = 0;
    }
```

10. Svc_Compl State

```
/* Extract data cell at head of the queue. */
```

```
/* This is the data cell just finishing service. */
pkptr_1 = op_subq_pk_remove (0, OPC_QPOS_HEAD);

/* Obtain original stream id from data cell. */
op_pk_nfd_get(pkptr_1,"Rsvd_ID",&stream_id);

/* Forward copies of the packet on all streams for which grants have */
/* been accepted, causing an immediate interrupt at each destination.*/
for(i=0;i<num_fixed_nodes;i++)
     {
     if((accept_grant[i]==1) && (status[i]==ACTIVE) &&
(xmit_hold[i]==0))
          {
          pkptr_copy = op_pk_copy(pkptr_1);
          op_pk_send_forced (pkptr_copy, i);
          }
     }
/* Server is idle again. */
server_busy = 0;

/* Decrement total cell count in queue. */
cell_queue = cell_queue - 1;
op_stat_write(cell_queue_stat,cell_queue);
/* Decrement total number of data cells in queue. */
data_cell = data_cell -1;

/* Destroy original packet. */
op_pk_destroy (pkptr_1);

/* If the queue is now empty of data cells and a grant is on hold, */
/* then clear the grant. */
if((data_cell==0) && (grant_hold==1))
     {
     grant_clear = 1;
     grant_hold = 0;
     }


/* Cancel Sent Request */
/* Reset flag_RM_immediate. */
flag_RM_immediate = 0;

/* If the queue is now empty of data cells and there are no */
/* outstanding requests to send data from other switches then */
/* cancel sent_request. */
if(data_cell==0)
     {
     canx_flag = 1;
     for(i=0;i<num_fixed_nodes;i++)
          {
          if(accept_request[i]==1)
               {
               canx_flag = 0;
               }
```

```
        }
    if(canx_flag==1)
        {
        for(i=0;i<num_fixed_nodes;i++)
            {
            sent_request[i] = 0;
            /* A new RM cell must be sent immediately from this */
            /* port.*/
            RM_port_flag[i] = 1;
            }
        /* If grant_hold has been activated, then the program will */
        /* transition to RM_xmit via Grant. Otherwise, it is */
        /* necessary for a transition to RM_xmit to occur so that */
        /* cancel of the sent request may immediately be sent to */
        /* the other switches. */
        if(grant_clear==0)
            {
            flag_RM_immediate = 1;
            }
        }

    /* If the queue is now empty of data cells and there is an */
    /* oustanding data request from one or more ports, then a new */
    /* grant must be generated to the appropriate port. */
    send_flag = 1;
    for(i=0;i<num_fixed_nodes;i++)
        {
        if(status[i]==ACTIVE && accept_grant[i]==0)
            {
            send_flag = 0;
            }
        }
    if(canx_flag==0 && grant_clear==0 && QUEUE_EMPTY==1 &&\
          send_flag==1 && grant_hold==0)
        {
        source_id[stream_id] = 0;
        grant_clear = 1;
        grant_hold = 0;
        }
    }
```

11. Svc_Start State

```
/* Schedule an interrupt for this process at the time where service
ends. */
op_intrpt_schedule_self (op_sim_time () + pk_svc_time, CELL_flag_xmit);

/*The server is now busy. */
server_busy = 1;
```

12. Idle State

```
immed_rcpt = 0;
```

LIST OF REFERENCES

[1] VADM A. K. Cebrowski and J. J. Garstka, "Network-Centric Warfare: Its Origins and Future," *The U.S. Naval Institute Proceedings*, pp. 28-35, Jan. 1998

[2] ADM A. Clemins, "IT-21: The Path to Information Superiority," http://www.chips.navy.mil/chips/archives/97_jul/file1.htm

[3] RADM R. M. Nutwell, "IT-21 Intranet Provides Big 'Reachbacks'," *The U.S. Naval Institute Proceedings*, pp. 36-38, Jan. 1998

[4] R. E. Parker, "Robust Transmission of Layered Video for Low-Bit-Rate Tactical Video Conferencing Applications," Doctoral Dissertation, Naval Postgraduate School, Monterey, California, September 1999.

[5] W. Stallings, *High-Speed Networks: TCP/IP and ATM Design Principles*, Upper Saddle River, NJ: Prentice-Hall, 1998.

[6] The ATM Forum, *User-Network Interface Specification Ver. 3.1*, Upper Saddle River, NJ: Prentice-Hall, 1994.

[7] R. Handel, M. N. Huber, and S. Schroder, *ATM Networks: Concepts, Protocols, Applications*, Harlow, England: Addison-Wesley, 1998.

[8] M. Schwartz, *Broadband Integrated Networks*, Upper Saddle River, NJ: Prentice-Hall, 1996.

[9] The ATM Forum Technical Committee, *Traffic Management Specification Version 4.0*, af-tm-0056.000, April 1996.

[10] G. Armitage, *Support for Multicast Over UNI 3.0/3.1 Based Networks*, Request for Comments 2022, November 1996.

[11] The ATM Forum Technical Committee, *Traffic Management Specification Version 3.1*, af-uni-0010.002, September 1994.

[12] S. Fahmy, R. Jain, S. Kalyanaraman, R. Goyal, B. Vandalore, and X. Cai, *Protocols and Open Issues in ATM Multpoint Communications*, http://www.cis.ohio-state.edu/~jain/papers/mcast.htm

[13] W. Stallings, *Data and Computer Communications*, Upper Saddle River, NJ: Prentice-Hall, 1997.

[14] M. Grossglauser, and K. Ramakrishnan, "SEAM: Scalable and Efficient ATM Multicast," *Proceedings of IEEE INFOCOM '97*, Vol. 2, pp. 867-875, April 1997.

[15] E. Gauthier, J. Le Boudec, and P. Oechslin, "Shared Access to Many-to-Many ATM Connections," *Transactions GLOBECOM '96*, Vol. 3, pp. 2123-2127, November 1996.

[16] E. Gauthier, J. Le Boudec, and P. Oechslin, "SMART: A Many-to-Many Multicast Protocol for ATM," *IEEE Journal for Selected Areas in Comms.*, Vol. 15, No. 3, pp. 458-472, April 1997.

[17] J. Gibson, T. Berger, T. Lookabaugh, D. Lindbergh, and R. Baker, *Digital Compression for Multimedia: Principles and Standards*, San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1998.

[18] R. Parker and M. Tummala, "Modeling of H.263 Encoded Low Bitrate Video Traffic for Tactical Video Conferencing Applications," *Proceedings of 32nd Asilomar Conference on Signals, Systems, and Computers*, Monterey, California, November 2-4, 1998.

[19] Recommendation I.363.2 - B-ISDN ATM Adaptation layer specification: Type 2 AAL, ITU-T, Geneva, Sep. 1997.

[20] P. Skelly, M. Schwartz, and M. Dixit, "A Histogram-Based Model for Video Traffic Behavior in an ATM Multiplexer," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 446-459, 8/1993.

[21] M. Riley and I. Richardson, *Digital Video Communications*, Norword, MA: Artech House, Inc., 1997.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

4. Prof. Murali Tummala, Code EC/Tu 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

5. Dr. Don Gingras...1
   SPAWAR Systems Center San Diego, Code D8805
   Communication and Information Systems Department
   San Diego, CA 92152-5001

6. Dr. Richard C. North..1
   SPAWAR Systems Center San Diego, Code D885
   53560 Hull Street
   San Diego, CA 92152-5001

7. LCDR Randolph R. Weekly, U.S. Navy 2
   315 Arloncourt Rd
   Seaside, CA 93955